2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-5388-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISSRE62328.2024.00039

Enhancing Black-box Compiler Option Fuzzing with LLM through Command Feedback

Taiyan Wang
CEE
NUDT
Hefei, China
wangty@nudt.edu.cn

Ruipeng Wang
CEE
NUDT
Hefei, China
wangruipeng@nudt.edu.cn

Yu Chen
CEE
NUDT
Hefei, China
cy@nudt.edu.cn

Lu Yu
CEE
NUDT
Hefei, China
yulu@nudt.edu.cn

Zulie Pan
CEE
NUDT
Hefei, China
panzulie17@nudt.edu.cn

Min Zhang*
CEE
NUDT
Hefei, China
zhangmindy@nudt.edu.cn

Huimin Ma
CEE
NUDT
Hefei, China
mahuimin17@nudt.edu.cn

Jinghua Zheng
CEE
NUDT
Hefei, China
zhengjinghua@nudt.edu.cn

***Abstract*—Since the compiler acts as a core component in software building, it is essential to ensure its availability and reliability through software testing and security analysis. Most research has focused on compiler robustness when compiling various test cases, while the reliability of compiler options lacks attention, especially since each option can activate a specific compiler function. Although some researchers have made efforts in testing it, the insufficient utilization of compiler command feedback messages leads to the poor efficiency, which hinders more diverse and in-depth testing.**

In this paper, we propose a novel solution to enhance black-box compiler option fuzzing by utilizing command feedback, such as error messages, standard output and compiled files, to guide the error fixing and option pruning via prompting large language models for suggestions. We have implemented the prototype and evaluated it on 4 versions of LLVM. Experiments show that our method significantly improves the detection of crashes, reduces false negatives, and even increase the success rate of compilation when compared to the baseline. To date, our method has identified hundreds of unique bugs, and 9 of them are previously unknown. Among these, 8 have been assigned CVE numbers, and 1 has been fixed following our report.

***Index Terms*—software testing, compiler options, fuzzing, large language model**

I. Introduction

The compiler serves as a pivotal tool in the software building process, which involves converting source code written in high-level programming languages into machine code that can be executed by the computer [1], [2], and deploying deep learning (DL) models on DL hardware [3]–[6]. Defects in compiler software can lead to compilation failures, significantly impacting the usability of programs [7]. Additionally, any inherent software defects within compilers can introduce potential security vulnerabilities into the compiled programs and influence the quality and functionality, thereby posing significant risks to the integrity and safety of the software supply chain. Consequently, both academia and industry have devoted considerable attention to enhancing the reliability and security of compiler software [8].

Compilers have two interfaces for software testing: ① The test cases to compile, which can be used to test for potential vulnerabilities when the compiler compiles different code under specific runtime scenarios. There are many researchers focusing on this topic, employing a range of techniques to generate diverse test case, including formal verification [9], [10], translation validation, differential testing [11], equivalence modulo inputs [12]–[14] and fuzzing [15]–[20]. ② The compiler options, which can be used to apply incremental code changes like optimizations during the compilation. Methods such as genetic algorithms [21] and fuzzing [22] are employed to generate different combinations of compilation command-line options, allowing for the testing of relevant functional code within the compiler.

Most research focuses on generating varied and well-formatted test cases to increase code coverage during fuzzing, but neglects the testing of compilation options. Each option enables specific functionality (e.g., optimization), thus providing greater diversity in testing. While there have been some attempts [21], [22] to use black-box fuzz testing, which utilize variations observed in the generated binary programs as feedback to guide the testing process, relying solely on the observation of such single final outcomes is insufficient for effectively exploring the space of compiler options combinations. In fact, the treasure is hidden within the command outputs of the compiler. We are inspired by the human strategy of adjusting compiler options during the compilation process, which involves modifying the combination of compiler options according to the command feedback outputs from the compiler. If the command feedback output of the compiler is utilized to guide the fuzzer, the fuzzer can generate a greater variety of legitimate compiler options combinations.

The command results of the compiler can be classified into

This work is supported by the National Key R & D Program of China, Grant Number 2021YFB3100500.

2332-6549/24/$31.00 ©2024 IEEE
DOI 10.1109/ISSRE62328.2024.00039

three categories: crash results (segmentation fault or some other core dumps) which serve as the target of the compiler fuzzing, compilation errors (error results, return code 1) which occur due to false option usages and are accompanied by error messages, and successful compilation (success results, return code 0) along with standard output. Current methods for fuzzing compilation options can achieve a certain scale coverage testing of the option search space and capture a large number of crashes. While in the mean time, a large amount of incompletely utilized error and success messages are obtained. Among the error results, there are some caused by targets of compilation with broken grammar, while a numerous test cases caused by using options incorrectly. By deleting options or modifying the parameter values, these errors can be eliminated to achieve successful compilation or directly converted into crashes. So are the success results, since some options forcibly output contents that may hide what should have been error or crash results. Deleting specific options can reduce time consumption and reveal more errors and crashes.

In this paper, we propose to enhance black-box compiler option fuzzing by leveraging command feedback, which includes command-line messages and compiled files. The command-line messages encompass error messages and standard output. To bridge the gap between an error and a prone crash, and to enable compiler to auto-fix running errors, we propose to modify the compiler command-line options according to the error messages. To dig errors and crashes hidden in the successful results, and to reduce running time, we propose to prune the options which force output to cover up crashes. Both methods above leverage intelligent semantic understanding technology, Large Language Model (LLM) to be more specific, and perform corresponding operations according to command feedback messages and option descriptions.

This paper aims to eliminate distracting information in black-box testing, enabling the triggering of more crashes while simultaneously increasing the success rate of compilation. By doing so, we can reduce the ineffective tests and reveal potential issues more accurately, thereby improving the efficiency and accuracy of the testing process. In summary, our contributions are as follows:

- We have thoroughly examined a significant number of false negatives among *error* and *success* results of existing compiler option fuzzing methods, and have uncovered crucial insights into how these false negatives occur. Our findings have highlighted the limitations of current methods and have inspired the development of our method.
- We propose a new method that leverages advanced LLMs to automate the process of *option fixing* and *option pruning*, thereby reducing false negatives and enhancing the efficiency of compiler option fuzzing.
- We conduct performance experiments and ablation study to evaluate our approach, and the results show that our approach enhances the detection of crashes, reduces false negatives, and even improves the success rate of compilation comparing to the baseline Cornucopia.

II. Background and Related Works

This paper targets the command-line options fuzzing of compilers, which is associated with two research areas: command-line options fuzzing for generic programs and compiler fuzzing. In this section, we first review related work for command line fuzzing and compiler fuzzing. Since we use LLM as a semantic understanding method and a decision maker in the fuzzing loop, the advanced fuzzing technologies assisted with LLM will be discussed.

A. Command Line Fuzzing

Current methods employ cluster analysis, white-box analysis of parsing option function code, and the use of information from option usage documentation, also design mutation methods for options, to achieve the goal of fuzzing for command line parameter options.

CrFuzz [23] targets the programs serving multiple purposes controlled by different options, leverages cluster analysis to notice whether inputs pass the specific parser. CLIFuzzer [24] extracts parameter specifications from "getopt()" functions, and mines option argument types through dynamic hooking analysis, then generates option predicates to fuzz. ConfigFuzz [25] encodes program options within part of the fuzzable input, so existing fuzzers can perform mutation to fuzz program configurations. CarpetFuzz [26] extracts conflicts and dependencies among program options from the documentation, according to the description of each option, and filters out invalid combinations to reduce the option combinations that need to be fuzzed. Existing methods are beneficial for our research on compiler option fuzzing, while they still struggle with handling the vast search space of compiler options in black-box testing, and they do not leverage command feedback as additional information.

B. Compiler Fuzzing

There are many researchers performing black-box, grey-box and white-box testing on compilers using various test cases, including those written in C and Go languages, among others. YARPGen [27] designs test-case generation policies and mechanism for avoiding undefined behaviors, to discover two kinds of bugs: internal compiler error and miscompilation. GrayC [28] generates compilable C programs with correct grammar, using coverage-guided fuzzing of a specific instrumented compiler, and tests on many other compilers/tools POLYGLOT [29] works on immediate representation (IR) level, It designs constrained mutation and semantic validation parts to mutate programs while preserving syntactic correctness and fixing semantic errors.

As for compiler option fuzzing, BinTuner [21] and Cornucopia [22] have employed this technique, but they conducted testing for different purposes. BinTuner [21] uses genetic algorithm to select command-line options of compilers for generating different binary variants, using normalized Compression Distance (NCD). Cornucopia [22] achieves feedback-guided compiler command-line options selection by using

Fig. 1. Motivation Example.

fuzzing, which is guided by maximizing the difference between binaries, and it aims to produce new binaries as well.

C. LLM Fuzzing

There is a growing trend towards applying LLM to fuzzing in many ways, like generating corpus and harness programs. Since LLMs are capable of generating format-correct test cases, they are effective for testing both APIs in deep-learning models and programs that accept programming language files.

TitanFuzz [30] is proposed to leverage LLMs trained on vast code corpuses to generate human-like input programs for fuzzing DL libraries, effectively enhancing code coverage and detecting previously unknown bugs. While in TitanFuzz, the LLMs often produce general programs similar to data in the pre-training process. To enhance the diversity of the corpus, FuzzGPT [31] leverages historical bug-triggering programs to synthesize unusual programs, thereby boosting fuzzing effectiveness.

As for fuzzing programs that accept programming language files, WhiteFox [32] proposes an analysis LLM to read source codes of compiler optimizations to summarize desired patterns of test cases, and proposes a generation LLM to produce test programs that trigger corresponding optimization codes. It is evaluated on the PyTorch Inductor [3], TensorFlow Lite [4] and TensorFlow-XLA [5], within PyTorch [33] and TensorFlow [34], to demonstrate its effectiveness. Fuzz4All [35] autoprompts LLM to generate test cases for nine kinds of SUTs that take six different programming languages as input, and achieves good results.

III. Motivation

A. Motivation Example

The goal of this study is to reduce false negative rate and to make fuzzing more effective and efficient. This section uses an illustrative example of a series of simple compiler option combinations to explain our motivation.

Fig. 1 shows a reduced option combination occurred in fuzzing. The option "–stackmap-version" in LLVM 18.1.2 with any value other than "3" will trigger the crash, since this option is used to specify the stackmap encoding version (defaults to "3"), while not all encoding methods are implemented correctly or have handled all exceptions.

However, the crash was not discovered until the combination (a) was analyzed which resulted in an error. When we first tried to repair the error associated with combination (a), it became evident from the error message feedback that the option "–mxcoff-roptr" should be deleted.

After deletion we got combination (b), while encountered another error message. We searched online to figure out that the problem existed in "–code-model" which carried wrong value "tiny", changing the value or deleting this option are both feasible solution.

Subsequently, we arrived at combination (c) but encountered a crash instead of a successful compilation, which means errors prevented the crash information from appearing. This

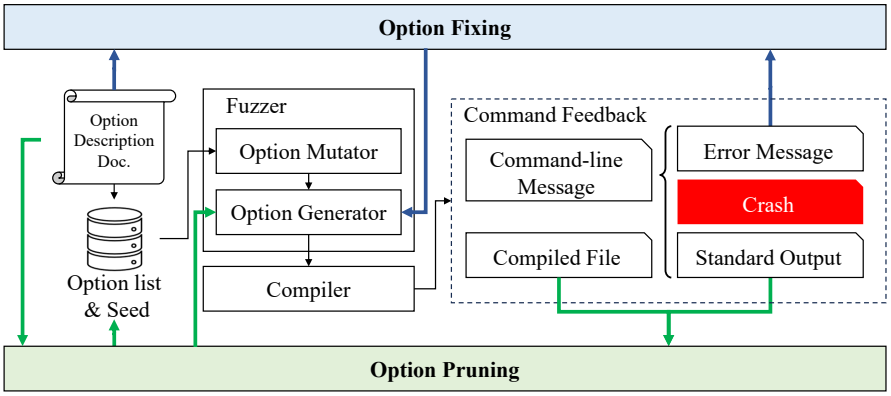

Fig. 2. The general workflow.

indicates that the results with return code 1 may have resulted in a crash after fixing the option combination, therefore reducing the false negative rates.

Apart from this, another observation is that when we added some options to print information, like "–version" in combination (d), the output information would forcibly print and overwrite crash messages. Consequently, some options that force the printing of messages may actually hinder our ability to identify bugs.

Although this example is limited to a small set of compiler options, the underlying phenomenon is widespread in the fuzzing results of existing methods. We have collected data during a fuzzing process, counted the different types of results as shown in Table I, and analyzed the options in both error and successful compilation cases. Among the error results of compilation, 37.58% (16,526 out of 43,978) indicate that they are caused by the redundant "–mxcoff-roptr" option, 48.82% (21,470 out of 43,978) are caused by incorrect value settings for the *code model* (related to the option "–code-model" and "–mcmodel") Besides 26.85% (24,404 out of 90,875) of successful compilations forcibly print version information without generating any compiled program, due to the presence of "–version".

In addition, there are other types of errors that can be fixed through minor adjustments, as well as related output options that provide different types of information from version information, but can impede the normal process of compilation or hinder the process of detecting deep vulnerabilities. These kind of compilation issues may be successfully resolved with further option adjustments, or they may uncover underlying bugs.

B. Our Solution

In black-box testing scenarios, where the internal workings of the system are not known, the test results (such as crash, error, or success flags) are the primary indicators. However, the command feedback, including command-line messages and the compiled files, can also guide the software testing process. Compiler command-line messages include error messages and standard output, both of which contain rich semantic information.

To address the issues mentioned before, we propose a fuzzing method that leverages LLM to semantically interpret command feedback from compilers, thus guiding the fuzzing process. This method consists of two newly added parts: *Option Fixing* and *Option Pruning*, which correspond to two different types of command-line output.

Given that error message is typically user-friendly and readable, it can directly assist users in achieving successful compilation. To eliminate the manual efforts in the workflow, we leverage the powerful semantic comprehension abilities of LLMs to understand this information and to generate suggestions for modifying compiler option combinations in order to fix errors.

The standard output in fuzzing results includes statistics on the running of compiler passes (modules performing the transformations and optimizations that make up the compiler) and so on, always accompanied by a heading that explains what the content represents, which can be used to predict the related compiler options. LLM is capable of understanding this information and can even retrieve additional data from online knowledge databases, thereby identifying options that can be pruned to minimize redundant output.

IV. Methodology

Fig. 2 presents the general workflow of our study, detailing the essential stages of our process. Our methodology starts with the generation of an option list from the option description documentation, the list includes all available options for compilers to use directly in fuzzing. The fuzzer leverages a corpus of options from the option list to generate combinations of command-line options, and any indicator is substituted with random data within the designated scale. Then the compilers under test are invoked with diverse option combinations and monitored whether any crashes occur, all the crash outputs along with error and successful results.

Apart from the general routine, we proposed two modules to reduce false negatives: *option fixing* and *option pruning*. The

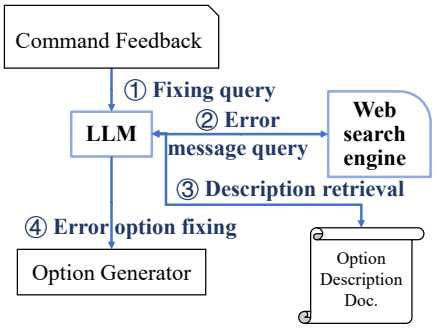

Fig. 3. Workflow of option fixing.

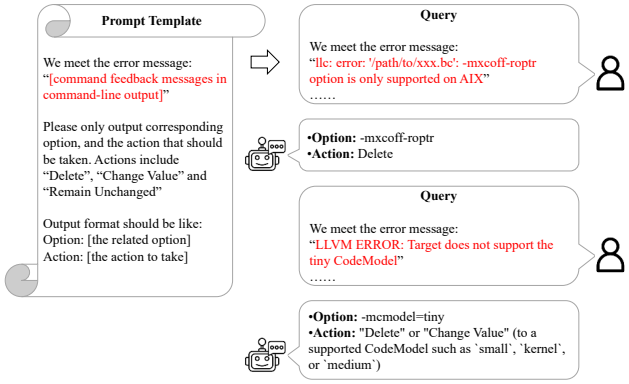

Fig. 4. Prompt and chat of fixing query.

option fixing module utilizes error messages from the command feedback produced by compiler testings. Our method builds upon the capabilities of LLMs, performs Retrieval-Augmented Generation (RAG) relying on web search engine and local compiler option description documentation, and finally provides suggestions on how to fix corresponding command-line option combinations to eliminate errors. The *option pruning* module utilizes standard output and even the compiled files, to determine whether there is any information that hinders the crash output or whether it is an invalid compilation resulting in an empty output file. We also employ LLM to align semantics between option description documentation and identify the specific option that controls the corresponding message printing by differential testing. By eliminating these options from the option list, we can reduce time consumption and potentially uncover hidden crashes.

A. Initial Fuzzing

Our approach begins with an initial fuzzing phase, which is instrumental in collecting critical information required for subsequent processes such as *option fixing* and *pruning*. This initial fuzzing stage includes all essential components of the workflow which consist of four distinct phases.

Firstly, we parse the option description documentation to generate an option list, which subsequently becomes our corpus. This documentation can be sourced from help pages and online documents. Options that do not require a value can be utilized by simply including or excluding them. If an option specifies a range of values, all possible value settings for that option should be enumerated and included in the option list. In cases where the range of values is too extensive to enumerate, we use indicators (such as labels like "$< int >$" or "$< str >$") to describe the value type.

Next, the option list is used to provide the fuzzer with all possible options, which the fuzzer then employs to generate various combinations of options. The seeds are also input into the fuzzer to be parsed into realistic options and then mutated based on the *option mutator* module. When the fuzzer encounters indicators, it parses a specific part of the random seed into byte strings for "$< str >$" and uses the modulus operation to constrain values within a specified range for "$< int >$".

Finally, the compilers are executed with the generated option combinations. All command feedback, including command-line messages and compiled files, is collected, and the running status is continuously monitored to ensure timely reporting of any crashes that occur.

B. Option Fixing

Following the initial fuzzing phase, we utilize the resulting test data, which comprises compiler command feedback messages and compiled output files, to assist in fixing compilation errors. The *option fixing* process works in parallel with the fuzzing loop, providing test cases with fixed option combinations for the fuzzer to execute. In this way, hidden crashes are discovered, and it can even achieve automatic compilation.

The *option fixing* process depends on two newly integrated components: a LLM that we query with test results to provide potential fixes, and a web search engine that the LLM uses to fetch online resources for improved accuracy. As Fig. 3 illustrates, the *option fixing* module consists of the four steps outlined below.

1) Fixing Query: First, test results that report errors are used to query the LLM about the causes of these errors and how to address them. To interact with the LLM, a prompt is necessary, and it is ideal for the prompt to be detailed, including examples and specific instructions. To obtain explicit and concise information that is easier for the program to process, it is recommended to establish a message format for the LLM to follow when providing responses. Additionally, providing examples of instructions and desired outputs will facilitate context learning.

Specifically, as shown in Fig. 4, the prompt is organized into three distinct components. The first part consists of command feedback messages, which provide context for the conversation. This is necessary because the option combinations generated by the fuzzer often contain too many tokens to be accommodated within the chat window when interacting with general LLMs. The second part comprises instructions that direct the LLM to identify which option is causing the error

message and to suggest a solution for the error. The third part establishes a standardized format for the information that the model should return, thereby facilitating automatic processing by the program.

It is worth noting that the third part is not necessary, because LLMs will format the output automatically when certain cues are given, such as "only output". Moreover, specifying an output format may restrict the model output, potentially limiting the diversity of responses and leading to an inefficient use of tokens.

To rectify errors, we have defined two primary candidate actions: "Delete" and "Change Value." The "Delete" action involves removing the problematic option entirely, while the "Change Value" action entails selecting a different appropriate value for the option. The "Add other options" action is not within the scope of our current actions, as incorporating new options would open up a vast new space for potential searches.

2) Error Message Query: In this step, the LLM utilizes a web search engine to search for additional articles related to the reported error message provided in the prompt. This functionality is supported by many commercial chat LLMs, such as GPT-4 [36] of OpenAI and Copilot [37] of Microsoft. By conducting real-time online searches, the LLM can gather more comprehensive information about the cause of the error and potential solutions. This additional information enhances the reliability of the LLM responses.

3) Description Retrieval: Following the online query about the error messages, the LLM obtains an initial understanding of the error causes and potential solutions, although this information requires further validation. The Retrieval Augmented Generation (RAG) is a technique employed to enhance the credibility of the LLM answers by referencing an authoritative knowledge base outside of its training data sources before providing a response. We consider the description documentation of compiler options to be such an external knowledge base. By consulting this documentation, we can more reliably identify the relevant option, and the detailed descriptions also helps to determine whether to delete the option or change it to a different value.

4) Error Option Fixing: Once the LLM has processed the query and retrieved relevant information from the description documentation, it generates strategies for fixing the options as shown in Fig. 4. The output includes the error-related option and the specific action to be taken, such as deletion or value change.

Subsequently, we apply the specified action to the identified option and rerun the compiler with the corrected options by manipulating the option generator. The hidden crashes and successful compilation are expected after deleting some option or changing the option value. If the error has not been , we will start an iterative process instead of one-time operation. The occurrence of hidden crashes and successful compilation is expected after deleting or changing an option value. If the error is not completely resolved, we will start an iterative process rather than ending with a one-time operation. If the error message remains unchanged, we initiate a three-layer

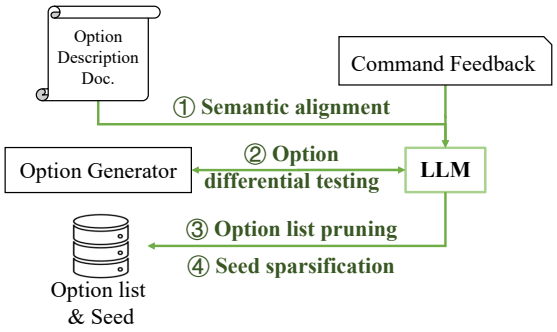

Fig. 5. Workflow of option pruning.

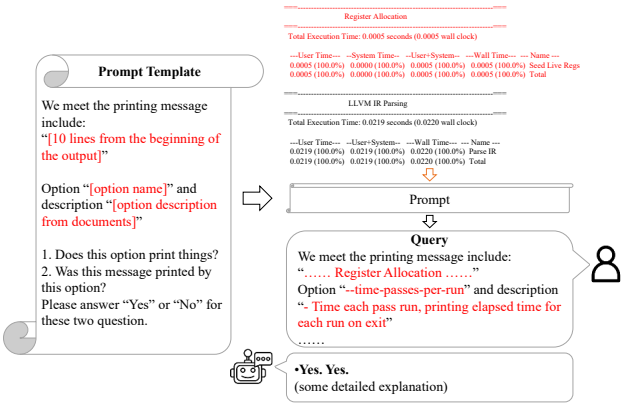

Fig. 6. Prompt and chat in option pruning.

loop prompting that involves repeating the query until either another error message occurs or the same error message is encountered for the third time. If another new error happens, a new three-layer loop prompting will be initiated for it. If the same error occurs three times, the prompting for this error will end, and the option fixing for next error case will begin. This iterative process is designed to address as many errors as possible within a limited time.

C. Option Pruning

The *option pruning* phase is conducted in preparation for the second round of fuzzing, aiming to increase efficiency by eliminating options that cause forced information output, thus reducing obscure crashes. The *option pruning* is performed in a serial mode before the next round of fuzzing, preparing an option list that will be used for generating test cases in the fuzzing process. This module is also built upon LLMs as shown in Fig. 5, and composed of four steps as follows.

1) Semantic Alignment: To prune irrelevant options, we first semantically align the options to identify those belonging to the output type. In this step, we utilize LLMs to analyze each line of the option description documentation, aiming to comprehend and determine whether an option is of the output type.

Meanwhile, we extract information from the beginning of the output messages(like ten lines for a balance between efficiency and information volume) that could indicate what kind of output are they, and align semantics between this information and the description documentation to identify the corresponding output options. These identified options will then be validated through subsequent differential testing. The prompt we designed and the ideal result that we achieved in chatting are depicted as Fig. 6.

2) Option Differential Testing: Differential testing begins with a crash Proof of Concept (PoC), which is a minimal option combination to cause a crash. Once we have identified options through semantic alignment, we verify each one by individually adding it to the crash PoC and checking whether it causes a crash or alters the status. If modifying the PoC does not crash, the newly added option is verified to hinder crash discovery and must be pruned.

3) Option List Pruning: To prune options, we simply need to remove them from the option list, as this list serves as the corpus for the fuzzer to generate test cases. Once all options that could hide crash reporting have been deleted, the second round of fuzzing will start.

Since there remains a possibility that the options to be pruned could be the cause of a crash, we must collect historical running status data to decide whether to remove them or to retain them for the potential to cause more crashes. If an option has been part of a combination that led to a crash in the past, it is advisable not to remove it, as it may potentially trigger additional crashes. Conversely, if an option has not been associated with any crashes, it is more suitable to remove it.

4) Seed Sparsification: Another related step in the workflow, along with option list pruning, is the sparsification of the seeds for the fuzzer. In compiler option fuzzing, seeds are simply random bytes that the fuzzer maps onto option combinations. However, seeds that are totally randomized without any pattern or structure awareness are clumsy, so we propose to sparsify the seed bytes into a new distribution space. This method involves reducing the range of seed values, making seeds be primarily composed of binary '0's and contain fewer binary '1's. This indicates that the number of options that are used in the generated option combination will be decreased, and it has been demonstrated to be effective to some degree in our experiments.

D. The Impact of Wrong Answers form LLMs

Since the output of LLMs involves some randomness, it is necessary to discuss what happens if we receive wrong answers in these two modules.

First of all, there are some common problems that both modules need to address, including the unexpected format of LLM output and non-existent options for operations. The output format may not follow the format we specified in the prompt, which can hinder our ability to extract content from a specific location. Therefore we often need to attempt multiple times and determine the appropriate session to use, with some human intervention required. As for non-existent options, they can be easily detected through string matching.

In *option fixing*, the answers from LLMs indicate the actions needed to deal with compilation errors, including whether to delete a specific option or change the value of a specific option. If the LLM provides a wrong answer, it can be either the wrong option to delete, or the wrong value for the option to replace. Actually instead of leading to a crash or success, wrong answers can result in two types of outcomes: the original error remains unchanged, or a new error is introduced. Since we have a three-layer loop prompting process as described in Section IV-B4, both old and new errors will be addressed during this iteration. The same old error has up to three chances for prompting and fixing, and each new error that occurs also triggers the same process. If the same error persists after three attempts at fixing it, the process ends. Crashes will be captured by the fuzzing framework, and successful compilations will be ignored in the fuzzing process. As a result, the time cost and computational resources spent on option fixing for a single error are limited, and the *option fixing* process is resilient to wrong LLM answers.

In *option pruning*, the LLM decides which options to prune. Incorrect results from the LLM may provide options that are irrelevant to the compiler output, and these will be handled by *option differential testing* as described in Section IV-C2. If an option passes *option differential testing*, we check historical crash information to see if it has ever been the cause of a known crash as detailed in Section IV-C3. Options that may potentially trigger vulnerabilities will be retained.

E. Implementation

The whole fuzzing framework is based on AFL++ [38], with the mutation module implemented by Cornucopia [22] through the use of custom mutators [39]. The code is written in Python, which is used for parsing option documentation into a list of available options and for generating randomized seeds.

The *option fixing* module is implemented in Python and operates on the option generator in fuzzer following each round of fuzzing. The *option pruning* module is also implemented in Python and is integrated into the fuzzing framework for offline processing, controlling seed generation and the option generator.

For the utilization of LLMs, we have different choices in the *option fixing* and the *option pruning* modules, due to varying demands on LLM capabilities. For *option fixing*, which requires the online retrieval capability, we choose the commercial Microsoft Copilot since it is powered by Bing online search engine. Additionally, given the need for document retrieval, we deploy the RAG service using the LangChain-Chatchat [40] project. For *option pruning*, which requires LLMs to achieve semantic alignment, we have tried open-source LLMs such as Vicuna [41], Alpaca [42], LLaMA [43] and RWKV [44]. Finally we found that the open-source ChatGLM3-6B based on GLM [45], [46] works fine with the RAG support provided by LangChain-Chatchat.

V. Evaluation

In this section we investigate the following research questions:

- **RQ1 (Effectiveness)**: What are the effects of our method?
- **RQ2 (Ablation Study)**: How does each component contribute to the effectiveness?
- **RQ3 (Bug Finding)**: Is our method able to detect real-world bugs?

A. Experimental Setup

Environment. All the experiments were conducted on a dedicated server with an Intel Xeon Gold 6230R CPU@104 × 4 GHz and 256 GB memory. We ran fuzzing in parallel against each combination of target and one test case, with identical configurations on 20 CPU cores for 2 hours.

Targets. We conducted our evaluation within the C/C++ compiler framework LLVM, focusing specifically on its internal toolchains, which include tools such as *clang*, *opt*, *llc*, *lli*, and others. This allows us to test LLVM comprehensively. We only target command-line options for fuzzing, thus we utilize a general set of benign test cases without grammar errors for compilation. This approach allows us to eliminate the influence of test cases and concentrate on the options.

Baselines. Since this paper is based on observations from the test results of Cornucopia [22], we select it as our baseline for comparison, and we also build our method based on the framework of Cornucopia. BinTuner [21] is another research effort that incorporates a module for testing compiler options. However it requires an explicit specification of conflicting compiler options and focuses on avoiding them, rather than aiming to discover crashes. Therefore we use Cornucopia as the sole baseline for comparison. Our method includes two newly added modules, the *option fixing* and the *option pruning*. The *option pruning* module also includes two parts: *seed sparsification* and *option list pruning*. Therefore in the ablation study we have eight versions of the method to demonstrate.

Metrics. We use the number of triggered crashes as our metric, since it is essentially the goal of fuzzing. The number of unique crashes is another metric for the evaluation, as the number of crashes with different root causes can reflect the code coverage to some degree in black-box testing.

B. Effectiveness

In our example presented in the Section III, we identified false negatives in both error and success test results in compilation, and our method aims at enhancing the results. To demonstrate the effectiveness of our method, we conducted a overall statistical analysis of the changes in error and success test results separately, and recorded the number of crashes.

Table I counts the test results of the baseline Cornucopia and our method, and compares the changes in the number of test results between them. It demonstrates that our method can discover significantly more crashes than the baseline, with an increase of 569.60%. The reduction in the number of error results indicates that our method effectively addresses errors. Although the number of successful results also decreases, the subsequent statistics will further validate the effectiveness of our method in achieving automatic successful compilation.

Table II illustrates that through our method, about 18.73% of success results obtained by Cornucopia were converted into crashes or errors that could not be handled, approximately 47.67% of error results were converted to successful compilation or crashes. These data confirm our observations and show the effectiveness of our work.

TABLE I
THE RESULTS OF GETTING DIFFERENT TEST RESULTS

Method	crash	error	success
Cornucopia	2434 (1.77%)	43978 (32.03%)	90875 (66.19%)
Our method	13864 (10.28%)	32556 (24.14%)	88433 (65.58%)
Change (Our/Cornucopia)	+ 469.60% (569.60%)	- 25.97% (74.03%)	- 2.69% (97.31%)

TABLE II
THE STATISTICS ON THE CONVERSION OF DIFFERENT TEST RESULTS

Type obtained by Cornucopia	From	Converted to		
		success	error	crash
success	90875 (100.00%)	73856 (81.27%)	9541 (10.50%)	7478 (8.23%)
error	43978 (100.00%)	14577 (33.15%)	23015 (52.33%)	6386 (14.52%)
crash	2434 (100.00%)			2434 (100.00%)

The number of unique crashes is another important metric for fuzzing evaluation. LLVM consistently throws a "Stack dump" when crash happens, which allows us to compare stack traces for classification after trimming the address information. We collected data in the conversion from success/error results to crash results and recorded this in Table III.

With our method, we observed that approximately 8.23% of the original success results and 14.52% of the original error results were converted into crashes, resulting in a significant increase in the total number of crashes. The number of unique crashes has increased to 245.36%, and even converting only success results contribute more unique crashes compared to original fuzzing method without our approach. Since our method processes test results from the baseline during the initial fuzzing period, the unique crashes we identify are able to encompass all types discovered by the baseline method.

TABLE III
THE NUMBER OF UNIQUE CRASHES WITH AND WITHOUT OUR METHOD

Type	number of crashes	number of unique crashes	
from success to crash	7478	215	476
from error to crash	6386	356	
original crash	2434	194	

C. Ablation Study

In the Section V-B, we chose a diverse set of general test cases for compilation, including gadgets from *coreutils* and other packages. In ablation study we only selected a single gadget, *augmatch* from the *augeas-tools* package, to make experiment smaller-scale and efficient.

The purpose of the ablation study is to investigate the individual contributions of each module to the overall results, and as stated in Section V-A we have eight versions of the method to compare: the baseline Cornucopia, the baseline with *seed sparsification*, the baseline with *option list pruning*, the baseline with both parts of *option pruning* and another four corresponding versions with *option fixing* module.

As listed in Table IV, we have recorded the total number of crashes and the number of unique crashes for each version of the implementation. The numbers in brackets represent the count of unique crashes.

The *option fixing* module acts more like icing on the cake instead of giving timely help, and the *option pruning* module contributes the most throughout the fuzzing in crash finding. The *option pruning* boost the result from 19 total crashes (6 unique) to 142 total crashes (20 unique), whereas the *option fixing* only achieve an increase from 19(6) to 44(8). However all four versions with the *option fixing* module in the lower section of Table IV still discovered more unique crashes than those in the upper section, demonstrating its effectiveness.

The *option list pruning* operation contributes the most in *option pruning* , consistently enhancing crash discovery capability. Nevertheless *seed sparsification* appears to be effective in limited circumstances, and even decrease from 19(6) to 5(3) when applied to the baseline only. In practical applications we only perform seed sparsification for half of the seeds to strike a balance.

D. Bug Finding

The Section III features a compact illustrative example. However, it is important to note that the actual trigger for the final crash is a command with a multitude of options, and the critical option (or options) responsible for the crash is obscured among the numerous available choices. To further investigate how our method uncovers a crash from a false negative test result and how we process the crash to determine whether it is unique, we will present a real-world case study from our experiments and show all real bugs we have analyzed.

1) Case Study: Following our methodology, we first conducted an initial fuzzing on the latest version of LLVM, and all command feedback messages were collected for subsequent analysis. We chose the command depicted in Fig. 7 as the starting point for our analysis.

All successful results with redundant printing messages will be processed by the *option pruning* module. Approximately 26.85% (24,404 out of 90,875) of the successful results (where the return code is 0) that we observed share the same command messages as those depicted in Fig. 7 (a).

Through *semantic alignment* period of the *option pruning* module, as depicted in Fig. 6, we automatically identified that

Fig. 7. A real-world case study.

the option "—version" is responsible for the messages. We selected one crash detected during the initial fuzzing phase, and added "–version" to the PoC command to see if the crash still occurred, employing a differential testing approach. We found the crash had disappeared, which proved that "–version" can obscure crashes. As a result, this option was removed from the list of options.

Following the offline processing of the *option pruning* module, our method proceeds to the second round of fuzzing using the updated option list. To remain faithful to the starting case, we extract the result after trimming "–version", transitioning from Fig. 7 (a) to an error result in Fig. 7 (b).

All error results will be processed by the *option fixing* module, and our analysis will advance accordingly. The error we have met here is "–mxcoff-roptr" again, as mentioned in Section III, and through interaction with LLM as shown in Fig. 4 we are able to fix this issue.

Subsequently, the compilation command executed successfully again, but it generated an empty compiled file along with the new command feedback message, as depicted in Fig. 7 (c). This suggests that there is still an issue with the command During the *semantic alignment* phase of the *option pruning* module, the output message was identified as being caused by the "–opt-bisect-limit" option. Consequently, it was removed from the option list. Therefore, we simply need to verify whether each option is included in the updated option list, or conduct another round of fuzzing, to eliminate the impact of this option.

Eventually we obtained a crash, as depicted in Fig. 7 (d).

After the fuzzing phase, which included periods of *option pruning* and *option fixing*, we finally obtained the command option combination that triggers a bug. However, as the combination includes more than six hundred options, we need to further investigate the critical options that are closely related

TABLE IV
THE STATISTICS OF CRASHES IN ABLATION STUDY

Type	initial	with seed sparsification	with option list pruning	with both
number of crashes (unique crashes) of baseline	19(6)	5(3)	66(20)	142(20)
number of crashes (unique crashes) of baseline with option fixing	44(8)	139(14)	847(36)	395(24)

to the crash.

It is often the case that only a few options, typically less than ten, are responsible for a crash. Therefore, we use the dichotomy method to identify these critical options from among the hundreds available, as described in Algorithm 1. We will split the options from $\mathbb{S}$ to X and Y using the dichotomy method, and then use X and Y individually to confirm whether they can trigger a crash. If one group triggers a crash and the other does not, the critical options are contained within the group that causes the crash. This allows us to reduce the scope of options to half of its current size with each iteration. If neither group of options triggers a crash when used independently after a certain round of splitting, it indicates that both groups contain critical options that are necessary to construct a crash PoC. Next, the two groups are each divided with dichotomy, and then tested recursively. The input parameter $\mathbb{I}$ is used to preserve one group of options while performing the dichotomy process on the other group. This process continues until the final irreducible result is obtained, which represents the minimum option combination PoC for triggering a crash.

2) Real Bugs: We have performed comprehensive testing on four versions (12.0.0, 14.0.0, 16.0.6, 18.1.2) of the LLVM framework, and the overall statistics of detected bugs are presented in Table VI. The total number of bugs we detected tends to increase with higher versions, which may be due to the increasing number of options from 1148 in LLVM-12.0.0, 2385 in LLVM-14.0.0, 2009 in LLVM-16.0.6, to 2594 in LLVM-18.1.2. Another possible reason is the lack of online support during the retrieval augmentation generation phase in the *option fixing* module.

All unique crashes are automatically calculated by scripts according to the stack dump, while all confirmed bugs are analyzed manually, keeping the count relatively low. We primarily focused on crashes in the latest LLVM-18.1.2 version and tried to apply the same PoC to the other, lower versions. All confirmed bugs were reported to the community and one of them has been fixed, hundreds of crashes are still pending analysis.

Eventually, 8 bugs were assigned CVE numbers, and another issue is still in progress. Each bug is associated with one component in LLVM, and the Table V lists the related tools, including *bugpoint*, *opt*, *llc*, *clang-repl*, *lli*, *clang-import-test* and so on. Each tool listed in the table has its own function, which involves optimizing similar files or converting one type of file format into another, like *bugpoint* helps to locate bugs by simplifying buggy bitcode file, and *opt* is used for optimizing LLVM Intermediate Representation (IR). As all of them belong to the LLVM framework, their functions and codes are coupled due to the shared use of several key components.

Algorithm 1 Algorithm of Dichotomy for Determining Critical Options

Input: Crash option set $\mathbb{S}$, Initial set $\mathbb{I} \leftarrow \varnothing$
Output: Critical option set $\mathbb{R}$

```
length ← size(𝕊)
half ← size(𝕊)/2
if half ≤ 1 then
    ℝ ← 𝕊
end if
for 0 ≤ i < half do
    X[i] ← 𝕊[i]
end for
for half ≤ j < length do
    Y[j-half] ← 𝕊[j]
end for
if Crash(X+𝕀) = 1 then
    if Crash(Y+𝕀) = 1 then
        Unexpected behavior !
    else if Crash(Y+𝕀) = 0 then
        ℝ ← Algorithm(X)
    end if
end if
if Crash(X+𝕀) ≠ 1 then
    if Crash(Y+𝕀) = 1 then
        ℝ ← Algorithm(Y)
    else if Crash(Y+𝕀) = 0 then
        ℝ ← Algorithm(X,Y) ∩ Algorithm(Y,X)
    end if
end if
```

VI. CONCLUSION AND FUTURE WORK

In this paper, we identified typical false negatives happened in current compiler option fuzzing methods, which include crashes obscured by error messages within the error results, and crash concealed by the forced standard output information within the success results. To address these two types of false negatives together, we designed an *option fixing* module and an *option pruning* module to enhance the existing black-box fuzzing, both methods utilized the intelligent semantic

TABLE V
THE DETAILS OF CONFIRMED BUGS AND RELATED TOOLS

CVE	bugpoint	opt	llc	clang-repl	lli	clang-import-test	llvm-tblgen	clang-refactor	llvm-xray	llvm-sim
CVE-2024-3xx22	+	+	+	+	+					
CVE-2024-3xx23	+	+	+	+	+	+	+			
CVE-2024-3xx24	+	+	+	+	+	+				
CVE-2024-3xx25	+	+	+	+	+	+				
CVE-2024-3xx28	•	•	•	•	•	•		•	•	•
CVE-2024-3xx29		+	+		+					
CVE-2024-3xx30		•	•	•		•				
CVE-2024-3xx31	+	+	+	+	+					
Issue#87053	+	+	+	+	+					

TABLE VI
THE STATISTICS OF BUGS DETECTED IN DIFFERENT VERSIONS OF LLVM

Version/Issue	Unique crash	Confirmed bugs	Pending	Fixed
LLVM-12.0.0	22	0	22	0
LLVM-14.0.0	131	3	128	0
LLVM-16.0.6	260	5	255	0
LLVM-18.1.2	476	9	467	1

understanding capabilities of the LLM. Through evaluation conducted under the limited but consistent computing environments and time constraints, our method significantly reduced the false negative rate, improved crash detection performance, and even discovered additional unique crashes. Many of these unique crashes have been reported to the community and some have been assigned CVE numbers.

Looking ahead, we plan to investigate other widely-used open-source compiler suites, such as the GNU Compiler Collection (GCC) [2], as our next research objective, in order to apply our method to a broader range of compilers. Additionally, another practical issue that requires addressing is the taxonomy of occurred crashes, which includes deduplication and identifying the smallest option combination that triggers each crash. Our method is relatively universal and can be applied to test other software, provided that it offers user-friendly command feedback and possesses a rich combination of command-line options for adjustment. It is feasible to conduct comprehensive compiler option testing in a white-box environment, as the corresponding source code offers a complete view of the program execution logic. This can be achieved by analyzing the option processing function "llvm::cl::ParseCommandLineOptions()" in a manner similar to how CLIFuzzer [24] analyzes "getopt()" functions.

ACKNOWLEDGMENT

This work is supported by the National Key R & D Program of China, Grant Number 2021YFB3100500.

REFERENCES

[1] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[2] GNU. (2024) Gcc, the gnu compiler collection. [Online]. Available: https://gcc.gnu.org/

[3] PyTorch. (2024) Torchinductor gpu profiling. [Online]. Available: https://pytorch.org/docs/stable/torch.compiler_inductor_profiling.html

[4] TensorFlow. (2024) Tensorflow lite — ml for mobile and edge devices. [Online]. Available: https://www.tensorflow.org/lite

[5] OpenXLA. (2024) Openxla project. [Online]. Available: https://openxla.org/xla

[6] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 03, pp. 708–727, mar 2021.

[7] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: How much does it matter?" *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[8] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. [Online]. Available: https://doi.org/10.1145/3363562

[9] M. A. Dave, "Compiler verification: a bibliography," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 6, pp. 2–2, 2003.

[10] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[12] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[13] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *Acm Sigplan Notices*, vol. 50, no. 10, pp. 386–399, 2015.

[14] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, 2016, pp. 849–863.

[15] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 95–105.

[16] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 445–458.

[17] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[18] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–26, 2022.

[19] H. Xu, Y. Wang, S. Fan, P. Xie, and A. Liu, "Dsmith: Compiler fuzzing through generative deep learning model with attention," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–9.

[20] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for javascript engines via deep compiler fuzzing," in *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*, 2021, pp. 435–450.

[21] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: an empirical study," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 142–157. [Online]. Available: https://doi.org/10.1145/3453483.3454035

[22] V. Singhal, A. A. Pillai, C. Saumya, M. Kulkarni, and A. Machiry, "Cornucopia: A framework for feedback guided generation of binaries," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3561152

[23] S. Song, C. Song, Y. Jang, and B. Lee, "Crfuzz: fuzzing multi-purpose programs through input validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 690–700. [Online]. Available: https://doi.org/10.1145/3368089.3409769

[24] A. Gupta, R. Gopinath, and A. Zeller, "Clifuzzer: mining grammars for command-line invocations," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1667–1671. [Online]. Available: https://doi.org/10.1145/3540250.3558918

[25] Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei, "Fuzzing configurations of program options," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1145/3580597

[26] D. Wang, Y. Li, Z. Zhang, and K. Chen, "CarpetFuzz: Automatic program option constraint extraction from documentation for fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1919–1936. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wang-dawei

[27] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428264

[28] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "Grayc: Greybox fuzzing of compilers and analysers for c," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1219–1231. [Online]. Available: https://doi.org/10.1145/3597926.3598130

[29] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz 'em all: Generic language processor testing with semantic validation," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 642–658.

[30] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[31] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623343

[32] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "White-box compiler fuzzing empowered by large language models," 2023.

[33] PyTorch. (2024) Pytorch. [Online]. Available: https://pytorch.org/

[34] TensorFlow. (2024) Tensorflow. [Online]. Available: https://www.tensorflow.org

[35] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the 46th International Conference on Software Engineering*, ser. ICSE '24, 2024.

[36] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[37] Microsoft. (2024) Microsoft copilot: Your everyday ai companion. [Online]. Available: https://copilot.microsoft.com/

[38] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[39] AFLplusplus. (2024) Custom mutators aflplusplus. [Online]. Available: https://aflplus.plus/docs/custom_mutators/

[40] chatchat space. (2024) chatchat-space/langchain-chatchat. [Online]. Available: https://github.com/chatchat-space/Langchain-Chatchat

[41] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," March 2023. [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/

[42] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford alpaca: An instruction-following llama model," https://github.com/tatsu-lab/stanford_alpaca, 2023.

[43] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[44] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV *et al.*, "Rwkv: Reinventing rnns for the transformer era," *arXiv preprint arXiv:2305.13048*, 2023.

[45] Z. Du, Y. Qian, X. Liu, M. Ding, J. Qiu, Z. Yang, and J. Tang, "Glm: General language model pretraining with autoregressive blank infilling," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 320–335.

[46] A. Zeng, X. Liu, Z. Du, Z. Wang, H. Lai, M. Ding, Z. Yang, Y. Xu, W. Zheng, X. Xia *et al.*, "Glm-130b: An open bilingual pre-trained model," *arXiv preprint arXiv:2210.02414*, 2022.