Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators

Kunpeng Zhang[1] Zongjie Li[1] Daoyuan Wu[1*] Shuai Wang[1*] Xin Xia[2]

[1] *The Hong Kong University of Science and Technology*
[2] *Zhejiang University*

Abstract

Modern software often accepts inputs with highly complex grammars. To conduct greybox fuzzing and uncover security bugs in such software, it is essential to generate inputs that conform to the software input grammar. However, this is a well-known challenging task because it requires a deep understanding of the grammar, which is often not available and hard to infer. Recent advances in large language models (LLMs) have shown that they can be used to synthesize high-quality natural language text and code that conforms to the grammar of a given input format. Nevertheless, LLMs are often incapable or too costly to generate *non-textual outputs*, such as images, videos, and PDF files. This limitation hinders the application of LLMs in grammar-aware fuzzing.

We present a novel approach to enabling grammar-aware fuzzing over non-textual inputs. We employ LLMs to synthesize and also mutate *input generators*, in the form of Python scripts, that generate data conforming to the grammar of a given input format. Then, non-textual data yielded by the input generators are further mutated by traditional fuzzers (AFL++) to explore the software input space effectively. Our approach, namely G^{2}FUZZ, features a hybrid strategy that combines a "holistic search" driven by LLMs and a "local search" driven by industrial quality fuzzers. Two key advantages are: (1) LLMs are good at synthesizing and mutating input generators and enabling jumping out of local optima, thus achieving a synergistic effect when combined with mutation-based fuzzers; (2) LLMs are less frequently invoked unless really needed, thus significantly reducing the cost of LLM usage. We have evaluated G^{2}FUZZ on a variety of input formats, including TIFF images, MP4 audios, and PDF files. The results show that G^{2}FUZZ outperforms SOTA tools such as AFL++, Fuzztruction, and FormatFuzzer in terms of code coverage and bug finding across most programs tested on three platforms: UNIFUZZ, FuzzBench, and MAGMA. G^{2}FUZZ also discovers 10 unique bugs in the latest real-world software, of which 3 are confirmed by CVE.

* Corresponding authors.

1 Introduction

Modern software often accepts inputs with highly complex grammars, such as images, configuration files, and network packets. Fuzzing such software is well-known to be challenging [14, 19, 23, 27, 33, 47] because it requires a deep understanding of software grammar to fully explore the input space. Often, one needs to prepare sample inputs that conform to the grammar of the input format and also exhibit a variety of characteristics before conducting effective greybox fuzzing [6, 15, 28, 35, 36, 48, 56] and uncover security bugs.

Recent structure-aware fuzzers have explored solutions to alleviate the above challenges with inference-based fuzzing and grammar-aware fuzzing, yet they still have limitations. Inference-based fuzzers, such as ProFuzzer [55], FuzzInMem [32], and WEIZZ [15], can infer input grammars and generate inputs on-the-fly. However, they often suffer from low accuracy and weak scalability, by inferring simple input fields and struggling to mutate file structures. ProFuzzer and WEIZZ are time-consuming for long inputs, while FuzzInMem requires programs having printer functions that convert in-memory data structures to files. Grammar-aware greybox fuzzers [5, 8, 21, 39, 51] often require pre-knowledge of the input grammar (e.g., provided by the users). Such information is often not available or incomplete, making it obscure to comprehensively understand input fields and their relations. Moreover, current fuzzing approaches primarily validate basic structural fields like size and checksums in file formats but neglect complex features. These features can trigger more complex logic, revealing deeper bugs. Complex features often include intricate chunks or constraints between chunks, posing challenges to traditional fuzzing methods. Fuzztruction [7] mitigates this challenge from a different perspective by injecting faults into generator applications to produce inputs with highly complex formats. However, Fuzztruction relies on the availability of a suitable generator application, which still requires experienced researchers to identify.

Large language models (LLMs) are transformer-based neural networks that have achieved state-of-the-art (SOTA) performance in natural language and code processing tasks. Thus, one may expect that LLMs could generate input samples with various valid grammars, thus driving grammar-aware fuzzing on its own. Indeed, we have seen some recent works that leverage LLMs to generate inputs for fuzzing [9, 10, 37, 53]. Nevertheless, we clarify that although LLMs are capable of generating *textual* outputs, such as natural language text and code, we find that LLMs are often incapable or too costly of generating *non-textual* data samples as required by many software. We present a detailed analysis in Sec. 3.

Instead of instructing LLMs to directly generate non-textual fuzzing inputs, this paper explores another perspective to augment mutation-based fuzzing with LLMs. The key idea is to leverage LLMs to automatically synthesize and further mutate input generators (often in the form of Python scripts) customized to the specific features and structures of the target file format. By executing these generators, we can produce inputs that exhibit a wide range of features and structures, potentially triggering different program logic and exploring previously untested code regions. Moreover, these generated non-textual inputs can be rapidly mutated by traditional mutation-based fuzzers, such as AFL++, to effectively explore the input space. Holistically, this approach offers a new and unique hybrid view to augment fuzzing with LLMs: *LLMs are particularly good at synthesizing distinct input generators and enabling the escape from "local optima," whereas mutation-based fuzzers excel at conducting fine-grained, local searches in the input space efficiently.* We show that our novel combination of LLMs and mutation-based fuzzers can achieve a synergistic effect, leading to a significant improvement in code coverage and bug finding. Moreover, since we only invoke LLMs when necessary to synthesize new input generators, we substantially reduce the cost of LLM usage.

We implement the above approach in a novel fuzzing framework, namely G^{2}FUZZ.[2] When users specify an input format name (e.g., "TIFF"), G^{2}FUZZ employs de facto LLMs, such as GPT-3.5 and llama-3-70b-instruct, to automatically synthesize input generators in Python scripts that generate TIFF images. G^{2}FUZZ facilitates several strategies to further mutate the synthesized generators. Then, G^{2}FUZZ executes generators to produce a diverse set of non-textual inputs, and also employs AFL++ to mutate the synthesized inputs. When the employed fuzzers fail to uncover new code coverage to a certain extent, G^{2}FUZZ invokes LLMs to synthesize new and distinct input generators, and then further mutates the generated non-textual inputs using AFL++. This process continues until the target software is fully covered or a certain time budget is reached.

We evaluate G^{2}FUZZ on 34 input formats, including JPEG images, TIFF images, and MP4 videos. Our results show that G^{2}FUZZ can consistently outperform SOTA mutation-based fuzzers, such as AFL++, and several structure-aware fuzzer baselines, in terms of code coverage and bug finding. We evaluated it on three third-party benchmarks: UNIFUZZ [30], FuzzBench [38], and MAGMA [22]. Our results show that G^{2}FUZZ achieves the best performance in code coverage and bug finding across all three platforms. We find that with the help of LLMs, G^{2}FUZZ is able to discover many unique edge/-function coverage that other fuzzers cannot find. Moreover, we show that G^{2}FUZZ incurs a very low cost of LLM usage; fuzzing a target software with GPT-3.5 for 24 hours only costs less than 0.2$ in LLM usage. We have used G^{2}FUZZ to find 10 unique bugs in the latest real-world software, of which 3 are confirmed by CVE. In sum, our contributions are as follows:

- We introduce a novel approach to augmenting mutation-based fuzzing using LLMs. The core idea is to combine the strengths of LLMs in synthesizing and mutating diverse input generators and the strengths of mutation-based fuzzers in performing fine-grained mutations over non-textual data. This approach leverages a synergistic effect to deliver effective fuzzing at a moderate cost.
- We design G^{2}FUZZ that concretizes the above idea. G^{2}FUZZ properly and periodically invokes LLMs and mutation-based fuzzers to benefit from their respective strengths. G^{2}FUZZ features a set of design principles and optimizations to make it highly efficient and practical.
- Our results show that G^{2}FUZZ consistently outperforms SOTA mutation-based fuzzers and several other fuzzer baselines in terms of code coverage and bug finding across various input formats and testing platforms. G^{2}FUZZ has discovered 10 unique bugs in the latest real-world software.

2 Preliminaries

Large Language Models (LLMs). LLMs, transformer-based neural networks, have reached SOTA performance in various NLP tasks, including translation and summarization. Autoregressive (e.g., GPT) and masked language modeling (e.g., BERT) are essential for textual output, while models like CLIP[29] and DALL-E[44] handle non-textual data, enhancing their range of applications. The community has noted that LLMs have the potential to augment software fuzzing [12].

Greybox Fuzzing. Greybox fuzzing, a technique for finding software security bugs, relies on lightweight instrumentation for execution feedback to mutate inputs more effectively. Fuzzers like AFL[57], AFL++[16], and Honggfuzz [2] have advanced this field. AFL++, with optimizations such as Redqueen, is recognized as the de facto standard fuzzer, widely used by the security community to detect bugs.

Grammar-Aware Fuzzing. Grammar-aware fuzzing, a form of greybox fuzzing, produces inputs based on precise grammar rules, effectively identifying vulnerabilities in software that handle complex input structures. Tools like FormatFuzzer [13], Gramatron [49], and Superion [51] leverage provided grammars to uncover security bugs in real-world soft-

[2] G^{2}FUZZ stands for "**g**rammar-aware fuzzing with LLM-synthesized input **g**enerators".

ware. To generate inputs in highly complex formats, Fuzztruction [7] deliberately injects faults into generator applications.

Inference-Based Fuzzing. Inference-based fuzzing, such as ProFuzzer [55], GreyOne [17], and WEIZZ [15], leverages inferred relationships between input bytes and path constraints to generate targeted test inputs. This method analyzes internal logic and data formats to create relevant test cases, enhancing coverage and reducing noise in results.

3 Motivation

3.1 Related Work and Limitations

Existing methods can be categorized into two types based on the input they handle: (1) Text-format fuzzing, and (2) Binary-format fuzzing. Text-format fuzzing primarily tests programs using text inputs, such as Superion [51], Nautilus [5], and Grimoire [8]. These methods generate a variety of valid text inputs based on provided specifications, including formats like XML, Ruby, SQL, and SMT. On the other hand, binary-format fuzzing tests programs with binary inputs, such as FormatFuzzer [13], FuzzInMem [32], WEIZZ [15], and AFLSmart [41]. These methods split the input into multiple chunks and perform mutations on these chunks to create diverse inputs, such as JPEG, PDF, TIFF, MP3, and MP4.

G^{2}FUZZ belongs to the latter category, constructing binary format files with complex features for exploring deeper into the code. Despite the significant advancements made by existing methods, they still face the following three challenges:

Challenge I: Generating Files with Complex Features. The current approaches focus primarily on the basic structure of target file formats, such as generating valid basic structural fields like size fields, checksums, and bitfields. However, a target binary file format often incorporates various complex features, and per our observation (see Sec. 5.1.1), files with complex features often have the potential to trigger more complex program logic, thereby likely uncovering deeper-seated bugs. Compared to the basic structures, these complex features differ mainly in two aspects: 1) complex features likely introduce extra complex chunks in the binary file, and 2) varying constraints (e.g., numerical constraints raised by checksum) may be introduced among binary file chunks. Additionally, certain dependencies exist among different (basic/-complex) features, where one feature depends on another to be implemented. For instance, to enable JPEG compression in a TIFF file, the file must first support the "YCbCr/RGB color space" feature. All these scenarios pose major challenges for existing binary-format fuzzers. For example, current fuzzers fail to generate TIFF files with LZW data due to inaccurate inference (e.g., WEIZZ) or incomplete grammars lacking LZW syntax (e.g., FormatFuzzer, AFLSmart), limiting their parsing and mutation capabilities. See Appendix A for more details.

Challenge II: Require Format Specifications and Manual Coding. Previous works often rely on provided format specifications or human effort to modify code, as seen in FormatFuzzer [13]. FormatFuzzer obtains format templates from the 010 Editor repository and uses them for parsing. However, manual coding is still required for the generation process. Furthermore, modifying complex formats like MP4 can take "over a week" (per [13]), due to its multiple chunk types, many of which are not fully detailed in the original binary template. Additionally, Fuzztruction can generate diverse files by injecting faults into generator applications. Yet, it relies on experienced researchers to manually identify and instrument suitable generator applications, and finding appropriate generators for less common formats is often challenging. Here, we search GitHub for generator applications for all 34 formats listed in Table 5. The search uses the keywords *"FORMAT converter/transformer/generator language:C++ stars:>5"*, where ""FORMAT" serves a placeholder for specific formats. For example, for JPG files, one of the search queries is *"JPG converter language:C++ stars:>5"*. Generators were found for 21 formats (usability untested), while no generators were available for the remaining 13 formats.

Challenge III: Simultaneously Process Multiple Formats. Many software can process multiple input formats. However, existing grammar-aware fuzzers typically generate files of a single format during the fuzzing process. This limitation hampers their effectiveness in thoroughly testing software that accept diverse input formats, potentially missing bugs related to the handling of specific file types.

One solution may be launching multiple fuzzers in parallel, each focusing on one input format, and then aggregating the individual fuzzing results at the end. However, programs often include routine code for preprocessing and error handling that are independent of specific file formats. Parallelism can result in repetitive efforts in these common routines, not only consuming time but also wasting resources.

Insight. We view that the aforementioned limitations can be addressed by cleverly leveraging LLMs. Our insights are as follows: for generating files with complex features (**Challenge I**), numerous libraries for file generation are already available online. These libraries offer APIs to directly construct complex features of the target format. Since LLMs have been trained on vast datasets that presumably include these online codebases, they shall be able to yield binary file generation scripts (code in Python) tailored to the required file features. By running this generator, we can produce files that exhibit the desired features. For example, to implement LZW compression for TIFF (Fig. 10b), we can employ LLMs to construct the corresponding generator with 3 lines of code, as in Fig. 11. With LLMs, common complex file structures can be generated with a moderate amount of code.

Using LLMs to generate generators evidently eliminates human efforts or the need for preparing format specifications. This enables a fully automated testing process, whereas existing methods require manual coding and format preparation (**Challenge II**). Moreover, our fuzzing pipeline maintains generators of different binary formats unifiedly (Sec. 4.1), allowing simultaneously processing multiple formats yet largely reducing repetitive efforts (**Challenge III**). This allows us to concentrate resources and efforts on more in-depth testing of code areas that are closely related to different file formats.

3.2 The Pilot Study of LLMs

LLMs have been extensively trained using large-scale datasets, enabling them to learn complex patterns and generate high-quality outputs. This extensive pre-training enables LLMs to excel in various open-ended, structured data generation tasks, such as code completion and generation [3, 11, 11, 20, 31, 40, 42, 50, 54, 58], text to image translation [4, 18, 24, 34, 43], and QA tasks for customer support [26, 45, 46]. It is believed that the vast amount of training data helps LLMs capture the nuances of language and produce accurate and contextually appropriate results.

Limitation of De Facto LLMs. We are positive that LLMs can be used to augment mutation-based fuzzing, given that LLMs may possess complex grammatical knowledge to facilitate continuous testing of software with complex input formats. However, we find that LLMs are often less capable or even incapable of generating *non-textual* outputs. In particular, while modern fuzz testing frequently targets non-textual inputs like image processing libraries, general-purpose LLMs are not designed to generate such non-textual data. Moreover, while cutting-edge LLMs such as DALL-E [1] can generate images, our tentative exploration shows that the image format is limited to a small set of predefined formats. DALL-E only supports generating images in common image formats, such as PNG and GIF, even if we specify requiring other formats (TIFF, RAW, BMP, etc.) in the prompts. Moreover, DALL-E usage costs $40.00 per 1,000 images, making it expensive for large-scale fuzzing. Generating an input sample can take several seconds, significantly impacting fuzzing throughput. Besides images, other non-textual file types, such as MP4 for video, MP3 for audio, PDF for documents, and Binary Large Object (BLOB), are often not supported by existing LLMs. Finding all customized LLMs can be challenging.

We analyze the input format distributions of FuzzBench programs. FuzzBench [38] is one most widely-used benchmarking platform developed by Google to evaluate fuzzing. It includes many widely used open-source projects that process a variety of input formats. We believe the analysis results will be generalizable due to the size and diversity of the benchmarks. We find that *73% of the programs only accept non-textual inputs.* Programs that accept non-textual inputs are more common than those accepting textual inputs in traditional fuzzing. For these programs, general LLMs cannot directly generate fuzzing inputs (reasons discussed before). Moreover, while some cutting-edge LLMs can generate PNG and JPEG inputs, other formats still lack support.

LLM-Enabled Opportunity. We observe that most binary files can be produced using Python libraries. For example, we can use PIL to generate JPEG files with different structures and CV2 to generate PNG files. Since documents for these libraries are likely included in the LLM training data, it is reasonable to expect that LLMs can synthesize generators for binary files based on these libraries. In this step, we conducted experiments on the available formats in UNIFUZZ, FuzzBench, and MAGMA to test whether LLMs can generate a generator for each format. For more details, refer to Sec. 5.1.5. In short, all these non-textual data can be generated by Python scripts. Consequently, we can use LLMs to synthesize various generators, producing different structured inputs and exploring deeper code regions. However, we have to address the following two challenges.

Technical Challenge I: Diversity. We find that LLM outputs are often less diverse and uneasy to control; this is undesirable in fuzzing, which expects a large number of generators that are diverse and cover as much of the input space as possible. Overall, our tentative exploration shows that LLM outputs are often predictable, meaning that the software under fuzzing may process many similar inputs that do not effectively cover the input space.

During our preliminary study, we attempted to calibrate the diversity of synthesized generators with several tactics, such as temperature control and top-k sampling, but the results were not satisfying. For example, while one may instruct LLMs to "*generate 100 JPEG image generators that are as diverse as possible*", we find that many of the generated images are simply repeated, and "100" is already too large for the LLM to process in one go. Recent works point out that using only LLMs to generate diverse samples is inherently challenging [25, 52]; this is often referred to as the "tail phenomena", where the LLMs tend to generate a large number of similar samples and only a small number of diverse samples.

Technical Challenge II: Overhead. Real-world fuzzing campaigns require many input samples to be generated and tested, and the suggested fuzzing duration is often in the order of days or weeks. This raises a severe concern on overhead. For example, the cost of using GPT-4 Turbo is estimated to be $10.00 per 1M tokens, and the cost of using DALL-E is estimated to be $40.00 per 1,000 images. Given that a single fuzzing campaign may require millions of tokens or images, the cost of using LLMs can be prohibitively high.

The time cost of using LLMs is also high, as the generation of a single input sample may take several seconds or up to twenty seconds, depending on the complexity of the input format and the quality of the generated sample. This is not practical for fuzzing, as the fuzzer is expected to generate and test a large number of input samples in a short time. Suppose each JPEG image takes 10 seconds to generate, and the fuzzer needs to generate 1,000,000 images. This will take 115 days to complete, which is not practical in real-world fuzzing.

4 Design

In line with challenges noted in Sec. 3, we present G^2FUZZ, a novel and efficient approach to augment mutation-based fuzzing with LLMs. Fig. 1 illustrates the high-level design of G^2FUZZ. G^2FUZZ features a hybrid strategy that combines a "holistic search" driven by LLMs and a "local search" driven by industrial-quality mutation-based fuzzers. The key idea is to leverage LLMs to automatically synthesize input generators that are customized to the specific features, structures, and grammar of the target file format. We further mutate the synthesized generators to enhance their diversity (see Sec. 4.1). By running these generators, G^2FUZZ can obtain seeds with diverse structures and features. Then, those generated inputs can be further mutated by traditional mutation-based fuzzers (AFL++) to explore the input space more effectively (see Sec. 4.2). When G^2FUZZ cannot identify a new path during the local search, it switches back to the holistic search to generate new input generators.

G^2FUZZ comprises two core components: *input generator synthesis* and *input generator mutation*. In input generator synthesis, G^2FUZZ first analyzes the file features for the target format and synthesizes an input generator for each feature. At this stage, we obtain some initial, rather simple generators. In the generator mutation stage, G^2FUZZ aims to produce generators customized to multiple features or structures simultaneously, which can yield more complex fuzzing inputs and enhance the generator diversity. We also evaluate the performance of each generator based on mutation feedback during fuzzing and extract useful knowledge from the successful generator mutations to guide future mutation directions. Except for specifying the target file format (which requires the user to provide), the whole process of G^2FUZZ is automated.

The bottom flowchart in Fig. 1 specifies the pipeline of standard mutation-based fuzzing. We start with initial seeds, add them to the seed queue, and then select a seed to mutate. Finally, we mutate the target seed under the predefined mutation strategy and check if the mutated inputs trigger bugs. G^2FUZZ augments the standard pipeline by incorporating the above two LLM-based components. Before seed selection, we obtain the fuzzing state based on the fuzzing performance. If it is the first cycle of fuzzing, we perform both input generator synthesis and input generator mutation to iterate basic features and enrich our initial seed corpus. If the fuzzing process cannot find a new path for a long time, G^2FUZZ directly mutates input generators, creating more complex generators through feature combinations and presumably enabling the fuzzing campaign to escape from "local optima."

Synergy Effect. We highlight that G^2FUZZ features a synergistic effect when combined with mutation-based fuzzers. LLMs are knowledgeable about the grammatical information of various input formats, yet they are less capable of generating those non-textual inputs directly. On the other hand, mutation-based fuzzers are good at performing fine-grained, byte-level mutations and exploring the input space systematically at low cost. However, conventional mutation-based fuzzers often lack the grammatical knowledge to generate high-quality input samples, and they often lack the "big picture" to progressively explore the input space. A good synergy between LLMs and mutation-based fuzzers can be achieved, where LLMs excel at synthesizing input generators and enabling escape from local optima, and mutation-based fuzzers excel at deeply exploring the local input space. This alleviates the limitations of both LLMs and conventional mutation-based fuzzers, and achieves better performance than using either of them alone; see evaluations in Sec. 5.

Addressing Technical Challenge I. Challenge I concerns the lack of diversity in LLM-generated outputs. As aforementioned, LLMs are inherently prone to the "tail phenomena" and often generate outputs that are repeated or very similar to each other. Rather than directly asking LLMs to produce "diverse" generators, G^2FUZZ first analyzes the possible features of a target file format, and then uses LLMs to synthesize input generators tailored to specific features/structures of the target file format. We also propose a set of strategies to extend and mutate the synthesized generators.

Addressing Technical Challenge II. Challenge II concerns the high cost of LLM usage. We address this challenge in a principled manner, where we only invoke LLMs to generate new input generators when needed. Holistically, LLMs are only invoked when the local search (conducted by AFL++) cannot identify new edges. This largely reduces the cost of LLM usage, from 15.16$ (our LLM-baseline setting; see comparisons in Sec. 5) to 0.124$, thus making G^2FUZZ practical and cost-friendly in real-world fuzzing campaigns.

Application Scope. G^2FUZZ is designed to be general-purpose and applicable to a wide range of input formats. We have evaluated G^2FUZZ on a variety of input formats, including JPEG images, TIFF images, MP4 videos, and 31 other formats. The design of G^2FUZZ is not specific to any particular input format, and it can be easily extended to support new input formats. With modern LLMs increasingly capable of gaining complex grammatical knowledge and coping with advanced data types (e.g., videos and audio), we are positive that G^2FUZZ can be used to augment mutation-based fuzzing for a wide range of input formats. We leave the exploration of those advanced data types to future work.

4.1 Input Generator Synthesis

When to Use. Before entering the formal fuzzing loop, G^2FUZZ first obtains the target input file format from the user (e.g., "TIFF"; this is the *only* information required), extracts its features, and synthesizes the corresponding generators. Then, it runs these generators previously-synthesized by the LLM to produce new diverse seeds and adds them to the seed queue. Note that if a software accepts multiple input file for-

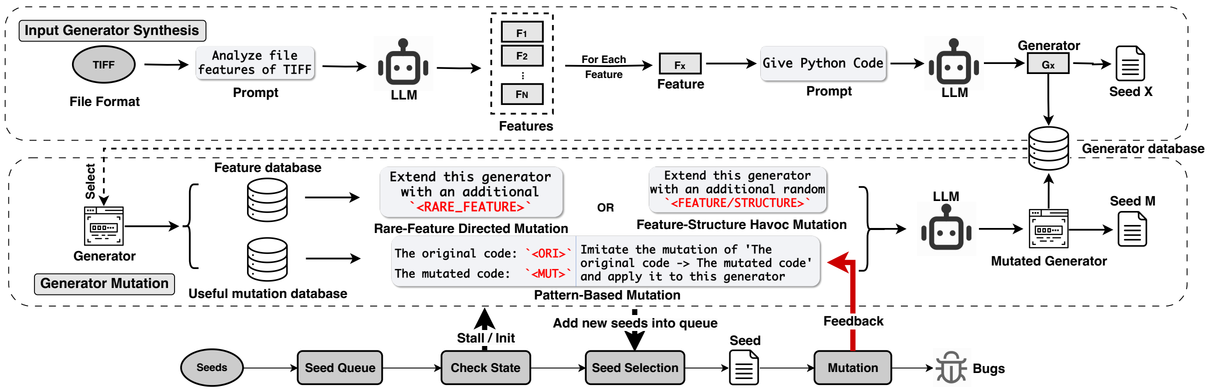


Figure 1: The workflow of G²FUZZ.

mats, G²FUZZ analyzes each format individually to obtain the corresponding generators.

Design Consideration: Features vs. Structures. To describe a file, two main aspects can be considered: *feature* and *structure*. "File feature" refers to attributes or characteristics that can provide external details about the file. "File structure" refers to the way data within a file is organized and formatted, providing internal details about how data is arranged within the file. Using file structure to describe a desired file input is a more straightforward approach. However, there is a gap between specifying structure and preparing the generator Python code. The document of Python file libraries often lacks details on how to write code to yield a specific file structure. In Python, constructing a file with a specific structure is not like building with blocks, where one chunk can be added at a time; instead, the file is often constructed from a more holistic perspective. This makes it difficult for an LLM to understand and use the libraries to achieve certain structures. Also, relations among chunks can be complex and have intricate dependencies. Our tentative study shows that creating generators based on structures has a high failure rate, consuming much time and negatively affecting fuzzing throughput.

We find that relevant Python file libraries often provide APIs to implement features for specific file formats, such as the compression flag in `libtiff` for storing TIFF files. In these cases, features and code have a direct map, as the document of these libraries includes corresponding descriptions. LLMs can learn from this information, making file features easy for them to understand. Consequently, the transformation from features to a Python generator code is straightforward. Indeed, since file features also encompass structural descriptions and complex constraints, generating input with a specific feature must adhere to the grammar and structural constraints. Therefore, we use file features to synthesize generators.

Overview. Input generator synthesis has two steps: given a file format, we first perform feature analysis to identify all possible features. Then, for each feature, we ask LLMs to synthesize a generator that produces an input with the target feature. As shown in Alg. 1, given a file format, G²FUZZ constructs a prompt and asks the LLM for the corresponding

Algorithm 1: Generator Procedure

Input: The target file format, *target_format*.
Output: A set of (generator, feature description), G.

1 $prompt \leftarrow construct\_prompt(target\_format)$ `// Based on Fig. 2`
2 $features \leftarrow LLM(prompt)$
3 **for** f *in features* **do**
4 $\quad g \leftarrow generator\_generation(target\_format, target\_feature)$ `// Based on Alg. 2`
5 $\quad$ **if** $g \neq none$ **then**
6 $\quad\quad seeds \leftarrow run(g)$
7 $\quad\quad add\_to\_queue(seeds)$
8 $\quad\quad G \leftarrow (g, f)$

features (lines 1-2). For each feature, G²FUZZ leverages an LLM to create a generator for it (lines 3-4). Then, G²FUZZ runs the generators to obtain seeds with various features and adds these seeds to the seed queue in fuzzing (lines 6-8).

Feature Analysis. As there are many feature descriptions in the document of Python file libraries, the LLM can synthesize a generator producing a file with the specific feature. To obtain the features for a given file format, we instruct the LLM to summarize the possible features. The prompt is shown in Fig. 2. For example, when applying this prompt to extract the features of TIFF files, the output might include: *1. Lossless compression: TIFF files support ... 2. Multiple layers: ....* We do not limit the number of features, as different file formats have varying ranges of features. We aim to capture common features at this step, and explore unusual features in Sec. 4.2.

```
What features can '<TARGET>' files have? Output the
information in the following format:
1. <feature 1>: <feature description>
2. <feature 2>: <feature description>
......
N. <feature N>: <feature description>
```

Figure 2: The prompt used to analyze the features of a specific program.

Generator Synthesis. After we obtain the features for a specific file format, we synthesize a generator for each feature. For the generator, we have two requirements: (i) it should be written in Python, and (ii) it should be executable. Obtaining a Python generator is straightforward for LLMs. However, there are several challenges for generators to run smoothly. First,

Algorithm 2: Generator Synthesis Algorithm

```
Input: The target file format, target_format. The target file feature,
    target_feature.
Output: A valid generator that can generate a file with specific features, g.
1  init_cnt ← 0
2  while init_cnt < INIT_MAX do
3      dialogue ← []
4      prompt ← construct_prompt(target_format, target_feature)
                                                   // Based on Fig. 3
5      dialogue.append(prompt)
6      g ← LLM(dialogue)
7      status, msg ← exec(g)
8      debug_cnt ← 0
9      while debug_cnt < DEBUG_MAX do
10         if status == SUCCESS then
11             ruturn g
12         error_info ← get_msg(msg)
13         while TRUE do
14             if "ModuleNotFoundError" not in error_info then
15                 break
16             library_prompt ← construct_prompt(error_info)
                                                   // Based on Fig. 4
17             relied_library ← LLM(library_prompt)
18             flag, g = automatic_installation(relied_library)
19             if flag == 0 then
                                     // Failed to install the library
20                 ruturn None
21             status, msg ← exec(g)
22             if status == SUCCESS then
23                 ruturn g
24             else
25                 error_info ← get_msg(msg)
26         dialogue.append(g)
27         dialogue.append(error_info + "Regenerate")
28         G ← LLM(dialogue)
29         status, msg ← exec(g)
30         debug_cnt ← debug_cnt + 1
31     init_cnt ← init_cnt + 1
```

as seed construction may rely on certain Python libraries, it is common to encounter the *ModuleNotFound* problem (Challenge I). Thus, we use the LLM to analyze the error information to automatically identify the required libraries and install them. Second, as LLMs cannot ensure the validity of generated codes, the code generated by an LLM may contain some bugs (Challenge II). We propose an algorithm to automatically debug the generated code.

As in Alg. 2, the synthesis algorithm first constructs an initial generator based on the target file format and the desired feature, and runs it to obtain the execution status (lines 3-7). The prompt template used is shown in Fig. 3. If the execution fails, the specific error message is extracted (line 12).

If the error involves a missing module, the algorithm attempts to install the required library. This process involves constructing a prompt based on the error information, using the LLM to identify the missing library, and then attempting an automated installation (lines 14-18). The prompt template is in Fig. 4. If the installation is successful, the generator is re-executed, and if the execution status is *SUCCESS*, the valid generator is returned (lines 22-23). Otherwise, the error handling loop continues.

After resolving the library dependency issue, the error information is fed back into the LLM to regenerate a program that can resolve the current error (lines 26-28). If debugging up to *DEBUG_MAX* times still fails to produce a valid generator, the algorithm attempts to generate a new initial generator (back to line 3). This is crucial because, due to the stochastic nature of LLMs, the same prompt can yield generators of varying quality, helping to avoid getting stuck in "local minima" (lines 9-12 and lines 26-30). Based on our preliminary exploration, we set *INIT_MAX* to 2 and *DEBUG_MAX* to 3 to balance the trade-off between the quality of the generated generator and the time cost.

~~~
Generate '<TARGET>' files containing the following features
using Python without any input files, and save the
generated files into './tmp/'.
```
<TARGET_FEATURES>
```
Please use Markdown syntax to represent code blocks. Please
ensure that there is only one code block. You don't need to
tell me which libraries need to be installed.
~~~

Figure 3: The prompt for developing a generator from a specific feature.

~~~
```
<MSG>
```
Please use Markdown syntax to represent the command. Please
ensure that there is only one command. To solve the above
issue using Python's package manager pip, you should run the
following command in the command-line interface:
~~~

Figure 4: The prompt for extracting the required library.

4.2 Generator Mutation

The generators obtained from Sec. 4.1 generate a seed with a single specific feature, which often covers a small part of the feature space. To effectively utilize the mutation feedback information from fuzzing and cover a larger feature space, we take into account more complex features using the following three mutation strategies: **Rare-Feature Directed Mutation:** We incorporate historical information—specifically, the features that have already been covered by the generators—into the prompt to guide the LLM in extracting unanalyzed features, focusing specifically on adding these rare features to the existing generators. **Feature-Structure Havoc Mutation:** We add a random feature/structure to the existing generators, aiming to unleash the potential upper bound capability of the LLM. **Pattern-Based Mutation:** As different features may exert varying influences on the target program, we leverage historical information to extract useful features and retain them by combining them with other features. Thus, we use the feedback information from the fuzzing process to guide the generator mutations.

When to Use. In the fuzzing process, generator mutation is executed in two specific situations. First, when G^{2}FUZZ initially

Algorithm 3: Generator Mutation Algorithm

```
   Input: The target file format, format.
   Output: A generator, g_m.
1  state ← get_fuzz_state()
2  if state == init then
3      prompt ← construct_prompt(target_format)  // Based on Fig. 5
4      features ← LLM(prompt)
5      for f in features do
6          g ← generator_select()
7          prompt ← construct_prompt(format, g, f)        // Based on
             Fig. 6
8          g_m ← LLM(prompt)
9          g_m ← self_debug(g_m) // Reuse the code lines 9 - 30 in
             Alg. 2
10         seeds ← run(g_m)
11         add_to_queue(seeds)
12 if state == stall then
13     g ← generator_select()
14     mutator ← mutator_choose()
15     if mutator == feature or mutator == structure then
16         prompt ← construct_prompt(format, g, mutator)  // Based on
             Fig. 13
17     else if mutator == pattern then
18         example ← pre_mutation_select()
19         prompt ← construct_prompt(g, example) // Based on Fig. 7
20     g_m ← LLM(prompt)
21     g_m ← self_debug(g_m)      // Reuse the code lines 9 - 30 in
         Alg. 2
22     seeds ← run(g_m)
23     add_to_queue(seeds)
```

enters the fuzzing loop, we use rare-feature directed mutation to enrich the variety of features in the initial seeds. Second, when fuzzing fails to find new paths within a set time limit (likely trapped in a local optimum), we use feature-structure havoc mutation and pattern-based mutation to construct seeds with different features or structures, which can help fuzzing explore other code regions.

Overview. When fuzzing stalls or initializes, G^{2}FUZZ uses generator mutation to generate more complex inputs. The algorithm is in Alg. 3. G^{2}FUZZ obtains the current fuzzing state (line 1). If it is initialization, G^{2}FUZZ performs rare-feature directed mutation. To do so, G^{2}FUZZ asks LLMs to extract the unanalyzed features based on historical information (lines 3-4). For each unanalyzed feature, G^{2}FUZZ randomly selects a generator and asks LLMs to incorporate the unanalyzed feature into it to create new inputs (lines 5-11).

If it is a stall, G^{2}FUZZ performs feature-structure havoc mutation or pattern-based mutation. G^{2}FUZZ randomly selects a generator from a database containing all executable generators (line 13). It then randomly chooses a mutator to apply to this selected generator (line 14). Next, G^{2}FUZZ constructs a prompt based on the chosen generator and mutator (lines 15-19). Finally, G^{2}FUZZ retrieves a mutated generator from the LLM, runs it to obtain a new seed, and adds this seed to the queue for further mutation (lines 20-23).

Rare-Feature Directed Mutation. To improve the comprehensiveness of our testing, it is essential to cover rare features that the generators from Sec. 4.1 may have overlooked. Our tentative study shows LLMs cannot often identify all relevant features of a file format in a single request. Typically, they provide around ten features at a time but often neglect rare features and cannot generate them directly.

To achieve rare feature mutation, we maintain a feature database that collects analyzed features as described in Sec. 4.1. Once a feature has been analyzed to synthesize a generator, its name, and corresponding description are added to the feature database. We then incorporate these analyzed features into a prompt and ask the LLM to identify other unexplored features, as illustrated in Fig. 5.

At the same time, we store all the synthesized generators in the generator database, and we randomly select a generator from this database. Afterward, we ask the LLM to mutate the selected generator to produce a file that includes an additional rare feature alongside the existing ones. The prompt for this step is shown in Fig. 6. Finally, we run the mutated generator, obtain new seeds with multiple (newly-added) features, and add this seed to the seed queue. Given that this method requires putting all previously analyzed feature descriptions into a prompt, we only use this strategy the first time fuzzing enters the loop to reduce token overhead.

~~~~
Analyzed features:
```
1. <feature 1>: <feature description>
2. ... : ...
```
Apart from the above features, what other features can '<TARGET>'
files have? Output the information in the following format:
1. <feature 1>: <feature description>
2. <feature 2>: <feature description>
...
N. <feature N>: <feature description>
~~~~

Figure 5: The prompt for rare feature extraction.

~~~~
```
<TARGET_GENERATOR>
```
The code above is used to generate <FROMAT> files. Now, we
need to extend this code to generate a new <FROMAT> file
that includes an additional `<NEW_FEATURE>` feature besides
the existing features. The description of the
`<NEW_FEATURE>` feature is as follows:
```
<FEATURE_DES>
```
~~~~

Figure 6: The prompt for rare feature mutation.

Feature-Structure Havoc Mutation. Although the LLM is powerful, its output can sometimes be unstable. To explore the full potential of the LLM, we ask it to randomly mutate the current generator to produce a file that includes an additional feature or structure alongside the existing features. The prompt is shown in Fig. 13. Since a generator can be mutated multiple times, it is possible to generate a file with many features or structures. The randomness of the LLM may introduce rare features that cannot be discovered through directed rare-feature mutation. While rare-feature directed mutation typically generates a file with two features, feature-structure havoc mutation can produce a file with more than two features.

This allows for the construction of more complex generators, enabling us to explore a deeper feature space.

Pattern-Based Mutation. Given that different features may exert varying influences on the target program, we propose pattern-based mutation. This approach uses historical information to extract useful features, which are then accentuated by integrating them with other features. Feedback from the fuzzing process effectively highlights which mutations result in more useful generators (i.e., those capable of discovering new edges). By analyzing this feedback, we can ask LLMs to learn these mutation strategies, thus guiding and optimizing future mutation directions.

The feature space of a file format is often vast, and not every unique feature triggers distinct processing logic in the target program. Iterating through all possibilities is inefficient; instead, we focus on the features that the target program is interested in, namely "program-relevant" features. Seeds with program-relevant features will be processed differently by the target program. If a seed obtained from a mutated generator discovers a new path, we infer that this seed contains program-relevant features. Consequently, we consider the $<$original generator, mutated generator$>$ tuple to contain useful information and incorporate it into our useful pattern database.

To reuse effective mutation strategies, we employ LLMs to learn the mutation patterns from the mutation generator tuples and apply these patterns to other generators. The prompt is shown in Fig. 7. By doing so, we aim to generate seeds with a richer variety of program-relevant features. In our implementation, we trace the performance of each generated seed during fuzzing. If a useful seed (i.e., one that discovers new paths) is produced through generator mutation, we consider this mutation useful for this program. Therefore, we add both the mutated and original generators to the prompt in Fig. 7 and apply this mutation strategy to other generator mutations.

````
The original code: ```<ORI>```
The mutated code:  ```<MUT>```
Imitate the mutation of 'The original generator -> The
mutated generator' above and apply it to the following
target code:
```
<TARGET_CODE>
```
````

Figure 7: The prompt for pattern-based mutation.

5 Evaluation

G^{2}Fuzz is built upon AFL++, enabling integration of our method with other existing techniques. To ensure the accuracy and fairness of our results, we conducted experiments on three testing platforms: UNIFUZZ, MAGMA, and FuzzBench. The experiments were carried out on three systems running Ubuntu 22.04, each equipped with 64 cores (Intel(R) Xeon(R) Gold 6444Y CPU) and 256GB memory. We study the following research questions (RQs): **RQ1:** Can the tool outperform SOTA in terms of code coverage and the number of unique bugs? **RQ2:** Can G^{2}Fuzz's performance surpass that of structure-aware fuzzers? **RQ3:** How many tokens will be consumed when fuzzing a program for 24 hours? **RQ4:** Which part of G^{2}Fuzz contributes the most?

5.1 Code Coverage and Unique Bugs

Code coverage and unique bugs are common metrics for evaluating fuzzers. To ensure fairness and reproducibility, we conduct our experiments on UNIFUZZ, MAGMA, and FuzzBench. All runtime settings, including the initial seeds, follow the default configuration.

5.1.1 Experiments on UNIFUZZ

UNIFUZZ is an open-source and metrics-driven platform designed for the holistic and fair evaluation of fuzzers.

Compared Fuzzers. AFL++'s performance demonstrates that the implementation significantly affects testing efficiency. To avoid the influence of implementation, we have developed G^{2}Fuzz based on AFL++, integrating it as a mode within AFL++. Since AFL++ already incorporates many SOTA fuzzers, this allows for easy and fair comparisons between G^{2}Fuzz and other fuzzers implemented in AFL++. As the range of features incorporated into AFL++ would exceed the scope of this paper, we compared G^{2}Fuzz with the four most widely used configurations of AFL++. **AFL++(cmplog)**: enables REDQUEEN mutator. **AFL++(mopt)**: enables MOPT mutator. **AFL++(fast)**: enables AFLFast seed scheduling. **AFL++(rare)**: prioritizes seeds that are rarely covered by other seeds. For G^{2}Fuzz, we enable cmplog mode to facilitate efficient low-level mutation.

Programs Selection. Since G^{2}Fuzz is designed for testing programs with non-textual inputs, we selected programs that meet this criterion. The target programs, listed in Table 17, include 10 programs with over 20 different input format types.

Experiment Results. Table 1 shows the edge coverage achieved by eight fuzzers. G^{2}Fuzz(GPT-4) discovers a total of 59,642 edges, which is 15,437 more than the best baseline fuzzer, AFL++(cmplog). G^{2}Fuzz(GPT-4) and G^{2}Fuzz(GPT-3.5) achieve the highest performance on 9 out of 10 programs, while AFL++(cmplog) excels on one program. Furthermore, G^{2}Fuzz(GPT-4) is able to discover more unique bugs than the other baseline fuzzers. Specifically, G^{2}Fuzz(GPT-4) finds 143 unique bugs, which is 32 more than the best-performing comparison fuzzer, AFL++(cmplog).

Moreover, we also calculated the pairwise unique code coverage. We summed up the unique code coverage for each pair of fuzzers across all programs, resulting in Fig. 8. G^{2}Fuzz(GPT-4) and G^{2}Fuzz(GPT-3.5) are quite similar, and both are able to identify more unique code coverage than the remaining four fuzzers. This result demonstrates that with the assistance of LLMs, the inputs we generated, which pos-

Table 1: Average code coverage and the total unique crashes found by G^2FUZZ with GPT-3.5/GPT-4 and 6 compared fuzzers.

Programs	**G^2FUZZ(GPT-3.5)**[1]		**G^2FUZZ(GPT-4)**[2]		**AFL++(cmplog)**		**AFL++(fast)**		**AFL++(mopt)**		**AFL++(rare)**	
	Cov.	#Bug	Cov.	#Bug	Cov.	#Bug	Cov.	#Bug	Cov.	#Bug	Cov.	#Bug
exiv2	5,099	**31**	**5,171**	28	4,965	26	3,776	5	3,758	12	3,851	11
ffmpeg	31,706	0	**34,218**	**1**	22,099	0	17,380	0	15,566	0	14,613	0
flvmeta	228	4	228	4	228	4	228	4	228	4	228	4
gdk	**2,958**	**6**	2,327	2	2,172	5	2,093	4	2,082	2	1,991	4
imginfo	2,839	0	**3,825**	0	2,189	0	1,998	0	2,004	0	1,976	0
jhead	315	13	**491**	**23**	445	21	195	3	195	3	195	4
mp3gain	921	11	921	**12**	**923**	11	900	10	899	8	891	10
mp42aac	2,067	**17**	**2,700**	14	2,091	13	1,178	6	1,157	3	1,135	2
pdftotext	**8,265**	43	7,921	**49**	7,434	25	6,483	25	6,376	28	11,257	28
tiffsplit	1,817	7	**1,840**	**10**	1,659	6	1,619	9	1,644	9	1,626	7

[1] **G^2FUZZ(GPT-3.5)**: G^2FUZZ based on GPT-3.5.
[2] **G^2FUZZ(GPT-4)**: G^2FUZZ based on GPT-4.

sess complex features, are capable of triggering more intricate program logic. Consequently, this leads to the discovery of a higher amount of unique code coverage.

G^2FUZZ incorporates additional steps into AFL++, such as generator synthesis and execution. To assess the impact of these steps on fuzzing throughput (i.e., execution speed), we measure the total number of executions for each fuzzer. Fig. 14 presents the total number of executions performed by all programs that the fuzzers run after 24 hours. The results indicate that G^2FUZZ's throughput does not significantly decrease compared to other fuzzers. We also fuzzed certain programs for 48 hours to observe G^2FUZZ's performance in the later stages of fuzzing. During the 24 to 48-hour period, G^2FUZZ discovered 32, 5, 1, 33, and 89 new edges in imginfo, jhead, mp3gain, mp42aac, and tiffsplit, respectively.

Furthermore, we evaluated the token costs of LLMs for fuzzing. G^2FUZZ minimizes token usage by reducing reliance on LLMs for mutation. GPT-3.5 incurs costs of less than 0.2$, and GPT-4 less than 13$ for 24 hours of fuzzing. More details can be found in Appendix B. Additionally, the ablation study shows that both components of G^2FUZZ are effective, with generator synthesis and mutation contributing 82,001 and 141,340 new paths, respectively. LLM-only approaches struggle, highlighting the necessity of integration. For more details, please refer to Appendix C.

5.1.2 Experiments on FuzzBench

To conduct more comprehensive experiments, we also evaluate G^2FUZZ using FuzzBench.

Experiment Setup. FuzzBench builds a Docker image for each fuzzer-benchmark pair to run the experiments. However, the resulting container does not have an internet connection. Therefore, we modify the condition to run the LLM-generation algorithm from 'stall/init' to 'init' to create a variant of G^2FUZZ. Specifically, we run the input generator synthesis and generator mutation when the fuzzing process first enters the fuzzing loop. This allows us to run the algorithm in advance and upload the results into the container built by FuzzBench. Consequently, we can conduct the experiments without needing an internet connection. To mitigate randomness in LLM generation, we conduct three epochs to

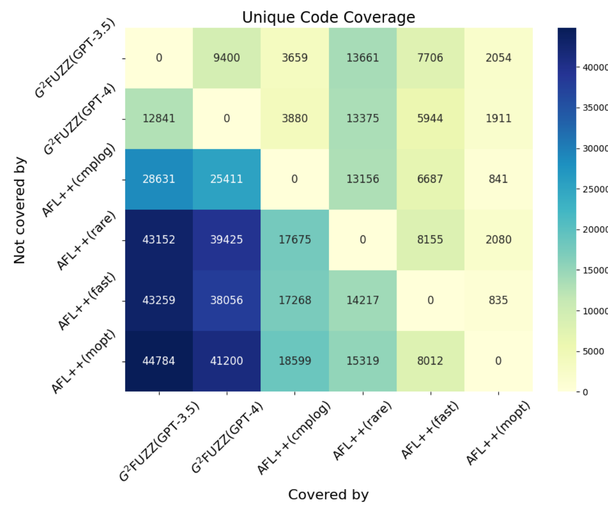


Figure 8: Pairwise unique code coverage across all programs. Each cell represents the number of code branches covered by the fuzzer of the column but not by the fuzzer of the row.

obtain three sets of seeds from the LLM-generation algorithm. Therefore, we need to perform three sets of experiments.

Programs Selection. Since G^2FUZZ is suitable only for non-textual inputs, we excluded programs with textual inputs, leaving 11 programs for testing G^2FUZZ, as listed in Table 17.

Metric. We choose the average rank of fuzzers as our evaluation metric to assess each fuzzer's performance across multiple benchmarks. By ranking the fuzzers on each benchmark according to their median reached code coverage, with lower values indicating better performance, we can derive an overall understanding of their effectiveness.

Experiment Results. As shown in Table 2, the performance of G^2FUZZ remains stable across different experimental groups. G^2FUZZ achieves the best ranks in all three groups, which are 2.09, 2.18, and 2.18. The second-best fuzzer, AFL++, achieves ranks of 2.73, 2.73, and 2.91 in the respective groups. In Figure 9, we additionally provide the code coverage distributions for all fuzzers across all programs within one of the experimental groups. [3] G^2FUZZ achieves the high-

[3] The report is available at: `https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/2024-05-16-formatfuzz/index.html`.

est performance on five out of 11 programs, while LibAFL and AFL++ each excel on two programs, and LibFuzzer and FairFuzz excel on one program each.

The LLM-generation algorithm demonstrates remarkable effectiveness on certain programs, such as *vorbis_decode_-fuzzer*. We illustrate the average code coverage evolution over time for *vorbis_decode_fuzzer* in Fig. 15. Notably, we observe that G^{2}FUZZ achieves higher coverage at 15 minutes compared to all other compared fuzzers at 23 hours. This highlights the capability of G^{2}FUZZ to generate diverse and complex structures, enabling the discovery of code regions that are challenging for conventional fuzzers to uncover.

Table 2: Fuzzbench fuzzer ranking. It reports the average rank of fuzzers, after we rank them on each benchmark according to their median reached code-coverage (lower is better).

Fuzzers	Group I	Group II	Group III	Average
G^{2}FUZZ	**2.09**	**2.18**	**2.18**	**2.15**
AFL++	2.73	2.73	2.91	2.79
LibAFL	4.55	4.55	4.64	4.58
LibFuzzer	4.73	4.64	4.64	4.67
HonggFuzz	6.45	6.45	6.36	6.42
AFLSmart	6.73	6.73	6.73	6.73
AFL	7.27	7.27	7.27	7.27
MOPT	7.27	7.27	7.27	7.27
Eclipser	7.36	7.36	7.27	7.3
FairFuzz	8.82	8.82	8.82	8.82
AFLFast	9.09	9.09	9.09	9.09
Centipede	9.18	9.18	9.18	9.18

Additional Analysis Time. Since we run the LLM-generation algorithm before the experiments, G^{2}FUZZ has more fuzzing time compared to other fuzzers. To assess its impact, we analyze additional analysis time per program, as shown in Table 18. The program with the highest additional time is *bloaty_fuzz_target* at 925 seconds (1.06% of 23 hours fuzzing time), while *zlib_zlib_uncompress_fuzzer* requires the least at 163 seconds (0.19%). Eight out of 11 programs need under 500 seconds (0.6%). We also find that the extra 15 minutes required for LLM-generation had no effect on median code coverage after 23 hours, as shown in Table 19.

5.1.3 Experiments on MAGMA

Code coverage and unique bugs are key metrics, but discovering real CVEs directly shows a fuzzer's ability to find security vulnerabilities with significant real-world impact. To avoid bias, we conduct experiments on MAGMA, a ground-truth fuzzing benchmark with real-world bugs for accurate performance evaluation. We integrate G^{2}FUZZ into MAGMA and compare it with five fuzzers (i.e., AFL++, MOPT, AFLFast, LibFuzzer, and Entropic). All AFL++-related fuzzers used in this experiment were based on AFL++ (commit 1d17210) and enabled the RedQueen mutator. We excluded programs with textual inputs. Table 17 shows the target programs used in MAGMA. Input format types are determined by file name extensions in the initial seeds. For *openssl*, as MAGMA's initial seeds lack extensions, we exclude it from consideration. To avoid randomness, we repeat the experiments 5 times.

Table 3: The total CVEs discovered (on MAGMA). G^{2}FUZZ performs the best in discovering real CVEs.

Program	G^{2}FUZZ	AFL++	MOPT	AFLFast	LibFuzzer	Entropic
libpng_read_fuzzer	**3**	**3**	2	1	**3**	0
tiff_read_rgba_fuzzer	**5**	**5**	**5**	4	2	3
tiffcp	**7**	6	4	4	0	0
pdf_fuzzer	**5**	**5**	4	2	2	2
pdftoppm	**6**	5	**6**	2	0	0
pdfimages	**6**	5	5	3	0	0

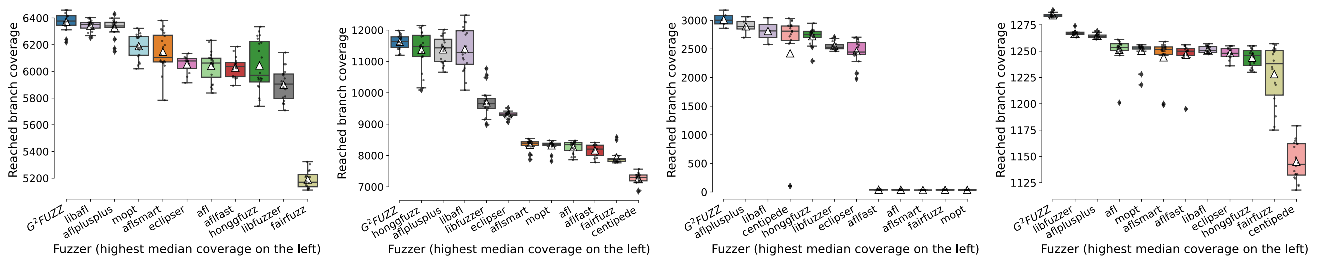
Table 4: The real bugs discovered by each fuzzer.

Program	G^{2}FUZZ	AFL++(cmplog)	AFL++(mopt)	AFL++(fast)	AFL++(rare)
mp3gain	1	1	-	1	1
pdftotext	1	1	1	1	1
mp42aac	**3**	**3**	1	-	-
mp42avc	**3**	-	-	-	1
mp42hevc	**2**	-	-	-	-
Total	**10**	5	2	2	3

We analyze the number of CVEs found by each fuzzer, whose results are in Table 3. We found that G^{2}FUZZ performs the best on the MAGMA benchmark, uncovering the most bugs in all programs. Specifically, G^{2}FUZZ performs the best on `libpng_read_fuzzer`, `tiff_read_rgba_fuzzer`, `tiffcp`, `pdf_fuzzer`, `pdftoppm` and `pdfimages`, exposing 3, 5, 7, 5, 6 and 6 bugs, respectively.

5.1.4 Finding Bugs in Latest Program Versions

To evaluate G^{2}FUZZ's ability to discover real bugs, we test the latest versions of projects from UNIFUZZ, along with all other executable programs in these projects suitable for fuzzing. Each fuzzer-program pair runs for 24 hours and is repeated 5 times. Following UNIFUZZ's suggestion, we use the top three functions from the ASAN output to de-duplicate uncovered bugs. The results are shown in the Table 4. G^{2}FUZZ discovers a total of 10 bugs, while the best comparative fuzzer, AFL++(cmplog), identifies 5. Notably, 4 bugs are exclusively discovered by G^{2}FUZZ and remain undetected by all other comparative fuzzers. By the time of writing, we have reported these bugs to the developers, and 3 of them have been confirmed by CVE: `CVE-2024-57509` (in `mp42avc`), `CVE-2024-57510` (in `mp42avc`), and `CVE-2024-57513` (in `mp42hevc`).



(a) bloaty (b) freetype2 (c) libpcap (d) vorbis

Figure 9: Code coverage distributions achieved in a FuzzBench experiment. Due to space constraints, we only present the results on four programs. For other experimental results, refer to Fig. 12.

5.1.5 Classification of Handled File Formats

In the above experiments across three platforms, we use G^2FUZZ to evaluate its effectiveness in handling various file formats. As shown in Table 5, G^2FUZZ successfully processes a variety of image formats including JPG, GIF, BMP, and PNG, as well as several audio and video formats such as MP3, WAV, MP4, and FLV. Additionally, G^2FUZZ demonstrates capability in handling PDF documents and various font formats like TTF and OTF. In terms of file formats, it supports processing formats such as ELF, Mach_O, and WebAssembly. All programs associated with the 34 formats were evaluated in the previous experiments. These findings highlight G^2FUZZ's strong performance in fuzz testing across diverse file types.

The conditions of constructing different file formats vary: some formats are supported by specific libraries, while others are not. We classify these conditions into three levels. (1) L1: The target format has specific libraries that can directly generate files. (2) L2: Some components of the format can be generated using existing libraries, and these components are then organized according to the target format's syntax rules. (3) L3: Files are generated entirely from scratch, based solely on the target format's syntax rules. Among the 34 tested formats, 23 fall into L1, 3 into L2, and 8 into L3. Formats in L1 typically lead to higher-quality generators the supporting libraries–often accompanied by documentation, sample code, and other resources–are included in LLM training data, increasing both the diversity and accuracy of generated files. Nevertheless, we also observe decent-quality generators for formats in L2 and L3, which demonstrates the robustness of G^2FUZZ in handling various formats.

Table 5: Classification of formats handled by G^2FUZZ.

Category	Level	Formats	Related Libraries
Image Formats	L1[1]	JPG	PIL/piexif
		GIF	PIL
		BMP	PIL
		PNG	PIL/matplotlib/cv2
		ICO	PIL
		XMP	lxml/xml.dom.minidom
		TGA	PIL
		TIFF	PIL/tifffile
	L2[2]	ANI	PIL
		RAS	PIL
		PGX	PIL
	L3[3]	PNM/RAW	-
Audio Formats	L1	OGG	soundfile
		MP3	pydub/mutagen
		WAV	wave/scipy
		AIFF	soundfile/pydub/wave
		AIFC	aifc
	L3	AU/CAF	-
Video Formats	L1	FLV	moviepy
		MP4	cv2/moviepy/mutagen
Document Formats	L1	PDF	fpdf/PyPDF2/reportlab
Font Formats	L1	TTF/OTF/WOFF/TTC	fontTools
File Formats	L1	Zlib compressed	zlib
		PCAP	scapy
		DER certificate	cryptography
	L3	ELF/Mach O/WebAssembly/ICC profile	-

[1] **L1**: Use specialized libraries to create files in the target format.
[2] **L2**: Construct parts of the file with specific libraries and organize them according to the target format's syntax rules.
[3] **L3**: Build the file from scratch based on the target format's rules, directly writing binary data or using *struct* to write the data.

5.2 Compared with Structure-Aware Fuzzers

We compare G^2FUZZ with the SOTA grammar-aware fuzzer FormatFuzzer and the SOTA inference-based fuzzer WEIZZ using the UNIFUZZ benchmark. We do not compare against AFLSmart because it has already been compared in FuzzBench, where G^2FUZZ significantly outperforms AFLSmart. G^2FUZZ's average code coverage rank is 2.15, while AFLSmart's is 6.73. Due to the considerable gap, we do not conduct additional experiments here. We do not compare against Superion, Nautilus, and Grimoire, as these fuzzers have only been assessed on text-based grammar input formats. Moreover, we do not include FuzzInMem, ProFuzzer, and GreyOne because they have not been made open source.

Table 6: The average line coverage discovered by G^2FUZZ, FormatFuzzer, and WEIZZ.

Programs	G^2FUZZ		FormatFuzzer		WEIZZ	
	line	function	line	function	line	function
exiv2	**5,984**	**1488**	1,138	369	3,732	1025
ffmpeg	**53,664**	**3028**	23,114	1554	26,789	1795
flvmeta	623	59	-[1]	-	**632**	**60**
imginfo	**5,003**	**364**	2,128	193	3,481	275
jhead	**431**	**21**	239	16	300	18
mp3gain	**2,168**	**58**	#[2]	#	2,103	56
mp42aac	**3,378**	**811**	#	#	2,041	504
pdftotext	**13,733**	**1182**	-	-	9,133	914
tiffsplit	**3,176**	**194**	-	-	3,019	185
gdk	**4,856**	**315**	2,287	192	=[3]	=

[1] **-**: FormatFuzzer does not support PDF, TIFF, and FLV formats.
[2] **#**: We encountered issues while running FormatFuzz.
[3] **=**: We are unable to compile *gdk-pixbuf* by WEIZZ.

Table 7: Functions exclusively discovered by each fuzzer.

Program	G^{2}FUZZ	FormatFuzzer	WEIZZ
gdk	**126**	0	-
exiv2	**424**	0	0
ffmpeg	**1951**	6	14
flvmeta	0	-	**1**
imginfo	**148**	0	0
jhead	**8**	0	0
mp3gain	**1**	-	0
mp42aac	**338**	-	0
pdftotext	**334**	-	0
tiffsplit	**10**	-	0

As G^{2}FUZZ, FormatFuzzer, and WEIZZ use different instrumentation methods, they may achieve varying edge coverage levels with identical inputs. To accurately measure line coverage, we utilize *afl-cov*. The results are presented in Table 6. To clarify, we encountered issues running some programs with FormatFuzzer and WEIZZ. Specifically, FormatFuzzer generated an excessive number of *core.** files while testing *mp42aac*, consuming over 500GB of memory within 10 hours. Additionally, we face errors when building MP3 generators for mp3gain according to FormatFuzzer's instructions. FormatFuzzer is also unsuitable for testing *pdftotext*, *tiffsplit*, and *flvmeta*, as it does not support PDF, TIFF, and FLV formats. As for WEIZZ, we are unable to compile *gdk-pixbuf*.

For nine out of 10 programs, G^{2}FUZZ achieves higher line coverage than both FormatFuzzer and WEIZZ. For example, G^{2}FUZZ achieves more than twice as much line coverage as FormatFuzzer in *exiv2*, *ffmpeg*, *imginfo*, *gdk*. Unlike the grammar-based fuzzer FormatFuzzer, G^{2}FUZZ is scalable to a broader range of programs that accept different formats. Moreover, it is common for a program to accept multiple input formats, but FormatFuzzer can handle only one format at a time, which can reduce the diversity of generated inputs.

To further validate that the files generated by G^{2}FUZZ with complex features help uncover more intricate program logic, we measure the number of functions exclusively discovered by each fuzzer. Here, "exclusive" refers to functions that are not detected by any other fuzzer. A fuzzer that finds more exclusive functions illustrates its ability to trigger more subtle program logics. The results are shown in the Table 7. For nine out of ten programs, G^{2}FUZZ identifies the largest number of exclusive functions, confirming the effectiveness of using LLMs to generate complex binary inputs.

5.3 Compared with Fuzztruction

To compare with Fuzztruction, we use G^{2}FUZZ to generate a batch of initial seeds and conduct experiments in the Docker environment provided by Fuzztruction, ensuring that all experimental parameters remain consistent. We test nine programs used by Fuzztruction; however, three lack clear input file extensions and are thus incompatible with G^{2}FUZZ. The tests run for 6 hours and are repeated 5 times, and the versions of all test programs are consistent with the versions tested by Fuzztruction. The mean coverage is shown in Table 8. Overall, G^{2}FUZZ outperforms Fuzztruction on seven out of nine programs. G^{2}FUZZ semantically constructs files with different structures from scratch, while Fuzztruction's generator–primarily a converter–still requires structured initial seeds, and its bit-level mutation only makes subtle adjustments. However, Fuzztruction performs better on programs using zip files. We believe G^{2}FUZZ finds fewer bugs due to the limited functionality of Python libraries for constructing zip files, restricting coverage of file characteristics.

We compare the seed quality of G^{2}FUZZ and Fuzztruction by measuring feature coverage. In this experiment, G^{2}FUZZ generates seeds only during the initial stage, while Fuzztruction continuously generates seeds. To ensure fairness, we compare the feature coverage of G^{2}FUZZ and Fuzztruction using the same number of generated seeds, with the number of seeds generated by G^{2}FUZZ serving as the baseline. For program selection, we target all programs that take image or document inputs, including pngtopng, pdftotext, and qpdf. In Fuzztruction, PDFs generated by the qpdf generator are excluded due to unparseable password options. The results are shown in the Table 9, revealing that G^{2}FUZZ discovers more unique features than Fuzztruction. In terms of validity ratio, G^{2}FUZZ achieves a higher validity ratio for PNGs and PDFs compared to Fuzztruction. Additionally, G^{2}FUZZ discovers more rare features, such as *Properties-Digital Signature* and *Properties-png:PLTE.number_colors* in PNG files, which Fuzztruction cannot cover.

Table 8: The average coverage (in basic blocks) and bugs discovered by Fuzztruction and G^{2}FUZZ.

Input Format	Program	Fuzztruction		G^{2}FUZZ	
		Cov	Bug	Cov	Bug
pdf	pdftotext-enc	36853.8	0	**38866.0**	0
pdf	pdftotext	35108.4	0	**39011.4**	0
elf	objdump	12468.8	0	**12851.8**	0
elf	readelf	12347.8	0	**13328.2**	0
png	pngtopng	4414.6	0	**4566.2**	0
der	vfychain	**14937.4**	0	11600.4	0
7z	7zip-enc	28887.2	**8**	**28909.0**	6
zip	7zip	**34585.4**	**8**	31691.4	6
zip	unzip	2788.2	1	**3104.8**	1

Table 9: Functions exclusively discovered by each fuzzer.

	PNG(pngtopng)		PDF(pdftotext)	
	Feature Cov	ValidNum/InvalidNum	Feature Cov	ValidNum/InvalidNum
G^{2}FUZZ	**457.0**	**36/0**	**531.0**	**50/12**
Fuzztruction	427.4	27.8/8.2	152.4	48.8/13.2

5.4 Feature Coverage

To verify whether G^{2}FUZZ can generate file features that other fuzzers cannot cover, we compare the feature coverage of each fuzzer. The calculation of feature coverage is challenging due to the lack of a unified quantification method. Therefore, we

Table 10: The feature coverage covered by each fuzzer.

Program	G²Fuzz	AFL++(cmplog)	AFL++(mopt)	AFL++(fast)	AFL++(rare)
TIFF (tiffsplit)	**4719**	2303	2169	2369	2178
JPG (exiv2)	**4096**	1910	1910	1913	1909
MP4 (mp42aac)	**1459**	0	0	0	0
PDF (pdftotext)	37797	33451	40730	**44704**	36203

use *ImageMagick* to extract each seed's attributes, such as compression type, treating each attribute as a feature. We then manually remove irrelevant attributes that vary across most files, such as the file name.

We select four formats–TIFF, JPG, MP4, and PDF–from Sec. 5.1.1 for analysis, covering image files, video files, and complex documents. The results are shown in the Table 10. G²FUZZ (using GPT-4) achieves the highest feature coverage for TIFF, JPG, and MP4. AFL++ focuses more on low-level mutations, which struggle to modify high-level features. Changing high-level features requires handling the constraints across multiple chunks simultaneously, which is highly challenging for byte-level mutation. In contrast, G²FUZZ can semantically mutate the generator or generate seeds with the target features from scratch, enabling broader coverage of high-level file features. Note that *ImageMagick* can only parse valid inputs, while most seeds generated by AFL++ mutations are invalid, resulting in lower feature coverage being recorded for AFL++. For example, the mutations of AFL++ fail to produce valid MP4 files, resulting in a feature coverage of 0. In the PDF format (*pdftotext*), AFL++ (rare) and AFL++ (fast) cover more features in terms of skewness, kurtosis, and standard deviation in the Blue/Green/Red channels. Seeds with such features receive higher weights in AFL++ (rare) and AFL++ (fast), leading to more frequent mutations and, consequently, higher coverage of these features. However, from the perspective of final code coverage, allocating excessive energy to explore such features is inefficient.

G²FUZZ can construct some rare features, such as *Properties-tiff:timestamp*, *Properties-tiff:copyright*, and *Properties-Contact* in TIFF files. Furthermore, for *Chromaticity-Compression*, G²FUZZ can cover all four compression methods–*Zip*, *RLE*, *JPEG*, and *LZW*–whereas other fuzzers can only cover *RLE* and *JPEG*. We observe that covering rare features can better trigger specific program logic in the target program, thereby improving code coverage.

5.5 Generalizability Across LLMs

To demonstrate G²FUZZ's generalizability across different LLMs, we select the open-source models *llama-3-8b-instruct* and *llama-3-70b-instruct* for our experiments. Under the same setup, these models generate initial seeds for five file formats, completing *Input Generator Synthesis* and *Generator Mutation* during the initialization stage. For *GPT-3.5* and *GPT-4*, we reuse the initial seeds generated from the first epoch of experiments for JPG, TIFF, MP3, MP4, and PDF with G²FUZZ on *exiv2*, *tiffsplit*, *mp3gain*, *mp42aac*, and *pdftotext*. Only files with the target format suffix are considered, as the generator may produce files in other formats.

The results are shown in the Table 11. *GPT-4* achieves the highest feature coverage for JPG and PDF, while the open-source models *llama-3-8b-instruct* and *llama-3-70b-instruct* achieve the highest feature coverage for TIFF and MP4, respectively. Notably, *llama-3-70b-instruct* outperforms *GPT-3.5* across all four formats. These results demonstrate G²FUZZ's scalability and its ability to generate high-quality generators using open-source models.

We also evaluate the effectiveness of the prompt used by G²FUZZ with 10 different formats (including images, videos, and documents) and find that GPT-4 performs well across most formats. More details can be found in Appendix D. Additionally, we analyze the impact of different libraries on the generator quality and find that collaboration between multiple libraries is the most efficient approach. For more details, please refer to Appendix E.

Table 11: Functions exclusively discovered by each fuzzer.

Format	GPT-3.5	GPT-4	llama-3-8b-instruct	llama-3-70b-instruct
JPG	259	**984**	211	636
MP4	-[1]	290	245	**517**
PDF	374	**559**	555	504
TIFF	388	387	**591**	516

[1] -: *GPT-3.5* cannot generate valid MP4 files during the first round due to randomness.

6 Discussion

G²FUZZ supports only file formats that have accompanied generator libraries available. Nevertheless, it can integrate with user-written file format specifications (by using prompts such as "*generate Python generator code based on the provided format specifications.*"). Thus, supporting corner cases or new file formats requires only extra engineering and manual effort. More importantly, we see an encouraging trend of emerging Python libraries for file generation (searching for JPEG and MP4 on GitHub reveals 21 and 26 file generation/editing libraries, respectively, created within the past three years); this illustrates the high extensibility of G²FUZZ to adapt to new formats without code changes. Overall, leveraging existing and emerging libraries, G²FUZZ can support more file formats, thus improving fuzzing efficiency in a broader range of scenarios and continuous manner. In addition, G²FUZZ is currently unable to handle custom formats. This limitation could be alleviated by adding document parsing capabilities, allowing LLMs to learn and adapt to custom syntax; we leave this as future work.

7 Conclusion

In this paper, we present G^{2}FUZZ, a novel and highly efficient approach that augments mutation-based fuzzing with LLMs. We identify a unique opportunity to combine the strengths of LLMs and mutation-based fuzzers to achieve a synergistic effect. The evaluation shows that G^{2}FUZZ consistently outperforms SOTA mutation-based fuzzers and several other fuzzer baselines.

8 Acknowledgment

We sincerely thank the anonymous reviewers and our shepherd for their valuable feedback and guidance. The HKUST authors are supported in part by a RGC GRF grant under the contract 16214723.

9 Ethics Considerations

Vulnerability Disclosure. Our fuzzing tool is designed to uncover security vulnerabilities in software. If we identify such vulnerabilities, failing to disclose them responsibly could lead to serious security risks, such as unauthorized access or exploitation by malicious actors.

We have established a responsible disclosure process. When we find vulnerabilities, we promptly notify the affected software vendors, giving them sufficient time to patch the issues before making any public disclosures. This ensures our research contributes positively to security without exposing users to unnecessary risks.

Experiments with Live Systems Without Informed Consent. If we apply our fuzzing tool to live systems or real-world software without obtaining consent from the owners or operators, this could disrupt services or negatively impact users who rely on those systems.

We avoid testing live systems without explicit consent from the system owners. When testing on live systems is necessary, we first obtain informed consent and design our testing methods to minimize any potential harm or disruption. This approach respects the rights and interests of those who rely on the systems we study.

Terms of Service. Our tool could potentially violate the terms of service of the software we are testing, particularly if the software explicitly prohibits automated testing or fuzzing. This could lead to legal issues or harm our reputation in the research community.

Before conducting any fuzzing, we thoroughly review the terms of service of the software being tested. If our activities might violate these terms, we seek permission from the software provider or adjust our methods to avoid violations. This ensures our research is both ethical and legally compliant.

Deception. If our testing involves any form of deception, such as obscuring the true nature of the tests from the system administrators or users, this could raise ethical concerns, particularly if it results in harm or a loss of trust.

We avoid using deception in our research. If deception is necessary for the validity of the study, it is ethically justified and followed by a thorough debriefing to explain the research's purpose and methods to all affected parties. This approach maintains transparency and trust.

Wellbeing for Team Members. Our work may expose team members to stressful or disturbing content, especially when analyzing malicious software, which could impact their psychological wellbeing.

We prioritize our team's wellbeing by supporting those exposed to stressful content, setting clear boundaries, and maintaining a safe, supportive work environment.

Innovations with Both Positive and Negative Potential Outcomes. The tools and techniques we develop have potential for misuse by adversaries. While our intention is to improve software security, there is a risk that others could use our tool to find and exploit vulnerabilities for malicious purposes.

We recognize the dual-use nature of our tool and have implemented safeguards to prevent misuse. Access is restricted, and we work with ethical review boards to assess risks. We also engage with the security community to ensure our research is used positively.

Retroactively Identifying Negative Outcomes. If our research unintentionally causes negative outcomes, like service disruptions or exploitation of vulnerabilities, failing to address them could harm users and damage our credibility.

We will monitor for any issues and take responsibility if they arise, working to remediate any harm. This proactive approach ensures our research remains ethical and responsible.

The Law. Our fuzzing activities must comply with cybersecurity laws and regulations. Any inadvertent violations could lead to legal consequences for us and our institution.

We consult legal experts to ensure compliance and obtain necessary approvals before engaging in risky activities, minimizing legal risks and ensuring proper conduct.

10 Open Science

To ensure compliance with open science principles, we commit to making our research data, code, and materials publicly accessible through publicly available repositories. This includes providing access to our tool code and experiment data at https://github.com/G2FUZZ/G2FUZZ[4] and https://github.com/G2FUZZ/G2FUZZ-DATA, allowing others to review, utilize, and adapt our implementation.

We also document our research methods, experiments, and results in detail to enable reproducibility. All relevant information will be shared openly to allow other researchers to replicate and build upon our work.

[4]Our tool code is also available at https://zenodo.org/records/14728879.

References

[1] Dall-e-3. https://openai.com/index/dall-e-3/.

[2] honggfuzz. https://github.com/google/honggfuzz.

[3] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. On hardware security bug code fixes by prompting large language models. *TIFS*, 2024.

[4] Maria Alabdulrahman, Renad Khayyat, Kawthar Almowallad, and Zahra Alharz. Sarid: Arabic storyteller using a fine-tuned llm and text-to-image generation. In *ICCAE*, 2024.

[5] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, 2019.

[7] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using fault injection-based fuzzing to leverage implicit domain knowledge. In *USENIX Security*, 2023.

[8] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. {GRIMOIRE}: Synthesizing structure while fuzzing. In *USENIX Security*, 2019.

[9] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *ISSTA*, 2023.

[10] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *ICSE*, 2024.

[11] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. Large language models of code fail at completing code with potential bugs. *NeurIPS*, 2024.

[12] Brendan Dolan-Gavitt. Is "ai" useful for fuzzing? (keynote). In *FUZZING Workshop*, 2024.

[13] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. Formatfuzzer: Effective fuzzing of binary file formats. *TOSEM*, 2023.

[14] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. Evolutionary grammar-based fuzzing. In *SSBSE*, 2020.

[15] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. Weizz: Automatic grey-box fuzzing for structured binary formats. In *ISSTA*, 2020.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *WOOT*, 2020.

[17] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. {GREYONE}: Data flow sensitive fuzzing. In *USENIX Security*, 2020.

[18] Hanan Gani, Shariq Farooq Bhat, Muzammal Naseer, Salman Khan, and Peter Wonka. Llm blueprint: Enabling text-to-image generation with complex and detailed prompts. *arXiv*, 2023.

[19] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215. ACM, 2008.

[20] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: A long-range pre-trained language model for code completion. In *ICML*, 2023.

[21] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *ICIA*, 2013.

[22] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *POMACS*, 2020.

[23] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *A-TEST*, 2018.

[24] Hui Huang, Shuangzhi Wu, Xinnian Liang, Bing Wang, Yanrui Shi, Peihao Wu, Muyun Yang, and Tiejun Zhao. Towards making the most of llm for translation quality estimation. In *NLPCC*, 2023.

[25] Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. Large language models struggle to learn long-tail knowledge. In *ICML*, 2023.

[26] Jaehyung Kim, Dongyoung Kim, and Yiming Yang. Learning to correct for qa reasoning with black-box llms. *arXiv*, 2024.

[27] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *SEN*, 2021.

[28] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *ASE*, 2018.

[29] Manling Li, Ruochen Xu, Shuohang Wang, Luowei Zhou, Xudong Lin, Chenguang Zhu, Michael Zeng, Heng Ji, and Shih-Fu Chang. Clip-event: Connecting text and images with event structures. In *CVPR*, pages 16420–16429, 2022.

[30] 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*, 2021.

[31] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *ASE*, 2020.

[32] Xuwei Liu, Wei You, Yapeng Ye, Zhuo Zhang, Jianjun Huang, and Xiangyu Zhang. Fuzzinmem: Fuzzing programs via in-memory structures. In *ICSE*, 2024.

[33] Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. Vd-guard: Dma guided fuzzing for hypervisor virtual device. In *ASE*, 2023.

[34] Yujie Lu, Xianjun Yang, Xiujun Li, Xin Eric Wang, and William Yang Wang. Llmscore: Unveiling the power of large language models in text-to-image synthesis evaluation. *NeurIPS*, 2024.

[35] 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*, 2019.

[36] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. Ems: History-driven mutation for coverage-based fuzzing. In *NDSS*, 2022.

[37] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *NDSS*, 2024.

[38] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *FSE*, 2021.

[39] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *SP*, 2020.

[40] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *SP*, 2023.

[41] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *TSE*, 2019.

[42] Jonathan Pilault, Raymond Li, Sandeep Subramanian, and Christopher Pal. On extractive and abstractive neural document summarization with transformer language models. In *EMNLP*, 2020.

[43] Leigang Qu, Haochuan Li, Tan Wang, Wenjie Wang, Yongqi Li, Liqiang Nie, and Tat-Seng Chua. Unified text-to-image generation and retrieval. *arXiv*, 2024.

[44] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *ICML*, 2021.

[45] Aryan Rangapur and Aman Rangapur. The battle of llms: A comparative study in conversational qa tasks. *arXiv*, 2024.

[46] Kuniaki Saito, Kihyuk Sohn, Chen-Yu Lee, and Yoshitaka Ushiku. Unsupervised llm adaptation for question answering. *arXiv*, 2024.

[47] Sevak Sargsyan, Shamil Kurmangaleev, Matevos Mehrabyan, Maksim Mishechkin, Tsolak Ghukasyan, and Sergey Asryan. Grammar-based fuzzing. In *IVMEM*, 2018.

[48] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. Fox: Coverage-guided fuzzing as online stochastic control. In *CCS*, 2024.

[49] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *ISSTA*, 2021.

[50] Liyan Tang, Zhaoyi Sun, Betina Idnay, Jordan G Nestor, Ali Soroush, Pierre A Elias, Ziyang Xu, Ying Ding, Greg Durrett, Justin F Rousseau, et al. Evaluating large language models on medical evidence summarization. *npj Digital Medicine*, 2023.

[51] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *ICSE*, 2019.

[52] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions. *arXiv*, 2022.

[53] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *ICSE*, 2024.

[54] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *MAPS*, 2022.

[55] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *SP*, 2019.

[56] 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 *USENIX Security*, 2020.

[57] Michal Zalewski. American fuzzy lop, 2017.

[58] Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B Hashimoto. Benchmarking large language models for news summarization. *TACL*, 2024.

A Challenges in Generating Complex Files: A TIFF Case Study

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
0x0000	49	49	2A	00	08	00	00	00	0A	00	00	01	04	00	01	00	00	00	64	00	00	00	01
0x0017	01	04	00	01	00	00	00	64	00	00	00	02	01	03	00	03	00	00	00	86	00	00	00
0x002E	03	01	03	00	01	00	00	00	01	00	00	00	06	01	03	00	01	00	00	00	02	00	00
0x0045	00	11	01	04	00	01	00	00	00	8C	00	00	00	15	01	03	00	01	00	00	00	03	00
0x005C	00	00	16	01	04	00	01	00	00	00	64	00	00	00	17	01	04	00	01	00	00	00	30
0x0073	75	00	00	1C	01	03	00	01	00	00	00	01	00	00	00	00	00	00	00	08	00	08	00
0x008A	08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

(a) Original TIFF file.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0x000	49	49	2A	00	1E	01	00	00	80	00	20	50	38	24	16	0D	07	84	42	61	50	B8	64	36	1D	0F	88	44	62	51	38	A4
0x020	56	2D	17	8C	46	63	51	B8	E4	76	3D	1F	90	48	64	52	39	24	96	4D	27	94	4A	65	52	B9	64	B6	5D	2F	98	4C
0x040	66	53	39	A4	D6	6D	37	9C	4E	67	53	B9	E4	F6	7D	3F	A0	50	68	54	3A	25	16	8D	47	A4	52	69	54	BA	65	36
0x060	9D	4F	A8	54	6A	55	3A	A5	56	AD	57	AC	56	6B	55	BA	E5	76	BD	5F	B0	58	6C	56	3B	25	96	CD	67	B4	5A	6D
0x080	56	BB	65	B6	DD	6F	B8	5C	6E	57	3B	A5	D6	ED	77	BC	5E	6F	57	BB	E5	F6	FD	7F	C0	60	70	58	3C	26	17	0D
0x0A0	87	C4	62	71	58	BC	66	37	1D	8F	C8	64	72	59	3C	A6	57	2D	97	CC	66	73	59	BC	E6	77	3D	9F	D0	68	74	5A
0x0C0	3D	26	97	4D	A7	D4	6A	75	5A	BD	66	B7	5D	AF	D8	6C	76	5B	3D	A6	D7	6D	B7	DC	6E	77	5B	BD	E6	F7	7D	BF
0x0E0	E0	70	78	5C	3E	27	17	8D	C7	E4	72	79	5C	BE	67	37	9D	CF	E8	74	7A	5D	3E	A7	57	AD	D7	EC	76	7B	5D	BE
0x100	E7	77	BD	DF	F0	78	7C	5E	3F	27	97	CD	E7	F4	7A	7D	5E	BF	67	B7	DD	EF	F8	7C	7E	5F	3F	A5	BA	02	0A	00
0x120	00	01	03	00	01	00	00	00	64	00	00	00	01	01	03	00	01	00	00	00	64	00	00	00	02	01	03	00	03	00	00	00
0x140	9C	01	00	00	03	01	03	00	01	00	00	00	05	00	00	00	06	01	03	00	01	00	00	00	02	00	00	00	11	01	04	00
0x160	01	00	00	00	08	00	00	00	15	01	03	00	01	00	00	00	03	00	00	00	16	01	03	00	01	00	00	00	64	00	00	00
0x180	17	01	04	00	01	00	00	00	16	01	00	00	1C	01	03	00	01	00	00	00	01	00	00	00	00	00	00	00	08	00	08	00
0x1A0	08	00																														

(b) TIFF file with LZW compression enabled.

Figure 10: Comparing two TIFF files with LZW compression enabled or not, both containing an identical image data. The newly added LZW-related chunks in Fig. 10b from (0x000, 0x08) to (0x100, 0x1D) cannot be parsed without specifications.

```
from PIL import Image
image = Image.new('RGB', (100, 100))
image.save('./tmp/tiff.tiff',
           compression='tiff_lzw')
```

Figure 11: Python generator for creating the TIFF file with LZW compression data in Fig. 10b.

In Sec. 3, we argue that generating files with complex features is challenging for current fuzzers. To illustrate this, we provide an example. TIFF, which stands for Tagged Image File Format, is a flexible and adaptable file format for storing images. Note that TIFF files support various compression algorithms. Here, we analyze the use of LZW compressed data within TIFF files to clarify why generating files with complex features is hard for existing fuzzers. Fig. 10 illustrates the differences between two TIFF files with an *identical* image data: Fig. 10a shows the original TIFF file, whereas Fig. 10b shows the file with LZW compression enabled. Two main differences exist: *1. introducing many (unparsed) data blocks.* In Fig. 10b, a large data block is introduced. Note that it is "unparsed" because the LZW specification is missing in the 010 Editor template used by FormatFuzzer, which prevents parsing and further mutating. *2. changes in data values and new constraints:* Many data values in Fig. 10b have changed, with these changes introduced new constraints (e.g., offsets and sizes) that need to be met. For example, when adding Exif features to a TIFF file, an ExifIFDPointer tag is added to the primary IFD to refer the Exif data. The Exif data, like color space, must be aligned with those in TIFF data, introducing new constraints between the Exif data and the primary IFD.

Based on our exploration, current binary-format fuzzers cannot generate TIFF files containing LZW data. These methods can be categorized into two types: *1. inference-based fuzzing*, such as WEIZZ. WEIZZ infers input fields and an approximate structure of the chunks on-the-fly while mutating. The inference results can be inaccurate, failing to precisely capture relations between chunks and making it unsuitable for constructing files with complex features. *2. grammar-aware fuzzing*, such as FormatFuzzer and AFLSmart. They rely on user-provided grammars to parse and mutate inputs. However, the standard TIFF specification can be insufficient frequently, and specifications of other formats are needed. In particular, due to the absence of LZW syntax in the grammar files shipped by FormatFuzzer and AFLSmart, they cannot generate TIFF files that include LZW data. Thus, even if an initial TIFF seed contains compression data, existing methods still cannot parse and mutate it.

Overall, complex features are very common across various file formats in different domains, such as the complex Exif data in JPEG files, transparency capabilities in PNGs, and encryption and DRM protection in MP4 files. It's worth noting that these complex features often involve more intricate logic and state management, which may likely result in security vulnerabilities. Therefore, constructing test input files with various complex features is crucial for enhancing fuzzing.

B Token and Cost Analysis

We further evaluate the token cost of LLMs for fuzzing. Overall, as we do not rely on LLMs to perform mutation engines, G^{2}FUZZ does not need too many tokens. We collect the token consumption of GPT-3.5 and GPT-4 in the experiments of UNIFUZZ. The results are shown in Table 12. In all programs, G^{2}FUZZ(GPT-3.5) costs less than 0.2$ for a 24 hour fuzzing process, while G^{2}FUZZ(GPT-4) costs less than 13$. We interpret that the token cost is acceptable for fuzzing.

Table 12: Token consumption and cost analysis for 24 hours of fuzzing in UNIFUZZ.

Programs	G^{2}FUZZ(GPT-3.5)		G^{2}FUZZ(GPT-4)	
	Token Count	Cost($)	Token Count	Cost($)
ffmpeg	57,870.0	0.10	112,763.8	3.97
gdk	80,720.4	0.14	186,810.8	6.50
jhead	68,816.2	0.12	158,385.6	5.68
mp42aac	64,593.6	0.12	170,129.4	6.03
tiffsplit	75,616.6	0.14	156,247.8	5.58
exiv2	41,958.4	0.08	100,312.0	3.58
flvmeta	52,277.4	0.09	350,607.6	12.71
imginfo	66,237.4	0.12	138,265.4	4.78
mp3gain	108,270.2	0.20	152,863.2	5.50
pdftotext	69,787.6	0.13	131,047.2	4.64

C Ablation Study

C.1 The contribution of each component

As G^{2}FUZZ comprises two main components: input generator synthesis and generator mutation, we analyze the contribution of each component. Our goal is to assess the effectiveness of the seeds generated by these components. If mutating a seed results in the discovery of a new path, we consider it useful. Therefore, we count the number of new paths found by mutating seeds from each component. The component that contributes more new paths is deemed more effective.

The results are presented in Table 13. On average, both the input generator synthesis and the generator mutation have proven to be effective. In total, the input generator synthesis contributes 82,001 new paths, while the generator mutation contributes 141,340. Specifically, in *jhead*, the input generator synthesis is responsible for discovering almost all the new paths. In *tiffsplit*, *ffmpeg*, *exiv2*, and *mp3gain*, the generator mutation contributes the most to discovering new paths.

Table 13: The number of new pathes contributed by the different components of G^{2}FUZZ.

Programs	Initial Seeds	Input Generator Synthesis	Generator Mutation
tiffsplit	101	539	2,549
jhead	2	1,046	0
mp42aac	6,859	522	6,298
gdk	9,832	12,993	1
ffmpeg	4,877	14,603	109,381
exiv2	398	40,719	18,328
flvmeta	1,133	1,066	29
imginfo	21,236	593	0
mp3gain	1,675	1,377	4,266
pdftotext	7,735	8,543	488
Total	53,848	82,001	141,340

C.2 Compared with LLM-Only G^{2}FUZZ

In G^{2}FUZZ, we leverage LLMs for generating diverse seeds and performing mutations using traditional byte-level techniques. Previous experiments confirm the LLM's effectiveness in seed generation. To assess the need for combining

LLMs with traditional methods, we created G^{2}FUZZ(LLM-Only), which solely relies on LLMs for seed mutation. Testing on UNIFUZZ shows that G^{2}FUZZ(LLM-Only) finds fewer edges and has lower throughput, often less than 1% of G^{2}FUZZ, as shown in Table 14. It also struggles with low-level mutations and is significantly more expensive, making the integration of LLMs and traditional fuzzing both necessary and efficient.

Table 14: The evaluation of G^{2}FUZZ(LLM-Only).

Programs	G^{2}FUZZ (LLM-Only)	Throughput	Token Count	Cost($)
flvmeta	150	5,758,008	16,253,249	15.72
exiv2	2,126	7,643	20,831,178	17.80
gdk	830	18,007	18,253,243	16.63
imginfo	909	13,155	17,353,735	16.10
jhead	245	150,634	18,523,648	17.00
mp42aac	728	11,645	16,349,895	15.13
tiffsplit	938	6,729	20,539,769	17.75
mp3gain	691	9,337	18,547,650	16.31
pdftotext	3,300	3,084	22,996,919	19.20

D Prompt Effectiveness

To evaluate the effectiveness of the prompts we used, we analyze three attributes. 1) Validity: The generator produced by G^{2}FUZZ should be able to construct valid seeds. 2) Proportion of Seeds with the Target Feature (PSTF): Seeds that contain the necessary code to produce the target feature are deemed to possess it. 3) Proportion of Unique, Useful Features (PUUF).

We analyze all generators from Sec. 5.1.1 for validity and manually review the generators produced during *Input Generator Synthesis* for the other two attributes. Specifically, we exclude files whose suffix matches the target format and use `ImageMagick` for analysis. Note that `ImageMagick` can process various image formats as well as PDF and MP4 (see Table 15). For example, for TIFF, we exclude seeds with a TIFF suffix from all programs, then parse each one with *ImageMagick*. Seeds that can be parsed are deemed valid, and vice versa. The results are shown in Table 15. GPT-4 achieves a validity rate exceeding 80% across all 10 formats, with PSTF over 70% in 5 formats and PUUF above 70% in 8 formats. These findings demonstrate the effectiveness of G^{2}FUZZ's prompts in efficiently accomplishing the target tasks.

The unsuccessful outcomes can be attributed to four reasons: 1) LLM hallucinations generate non-existent features. 2) Debugging (Alg. 2) leads the LLM to remove code related to the target feature for proper execution. 3) Rare features are harder to generate. 4) Some features exist in all files of a given type, rendering them useless. We further analyze the impact of LLM hallucinations on G^{2}FUZZ. During the feature generation phase, hallucinations are relatively rare, with most useless features like "default features describing the target format" or "redundant features." Hallucinations primarily occur during generator synthesis, often referencing non-existent functions or attributes and triggering exceptions such as *AttributeError* or *NotImplementedError*. However, due to our debugging strategy (Alg. 2), these errors caused by hallucinations are promptly detected when executing the generator. The LLM then attempts to fix them, effectively mitigating the impact of hallucinations.

Table 15: Analysis of prompt effectiveness for different formats.

Format	Validity Rate GPT-3.5	Validity Rate GPT-4	PSTF (GPT-4)	PUUF (GPT-4)
TIFF	91.90%	94.31%	90.00%	90.00%
BMP	57.25%	97.26%	50.00%	60.00%
JPG	98.05%	99.16%	70.00%	80.00%
PNG	90.78%	99.65%	77.77%	77.77%
GIF	88.37%	100%	75.00%	75.00%
ICO	47.88%	100%	57.14%	85.71%
TGA	22.22%	82.81%	88.88%	88.88%
PNM	67.12%	90.00%	37.50%	37.50%
MP4	35.29%	90.17%	60.00%	80.00%
PDF	98.80%	95.71%	60.00%	86.66%

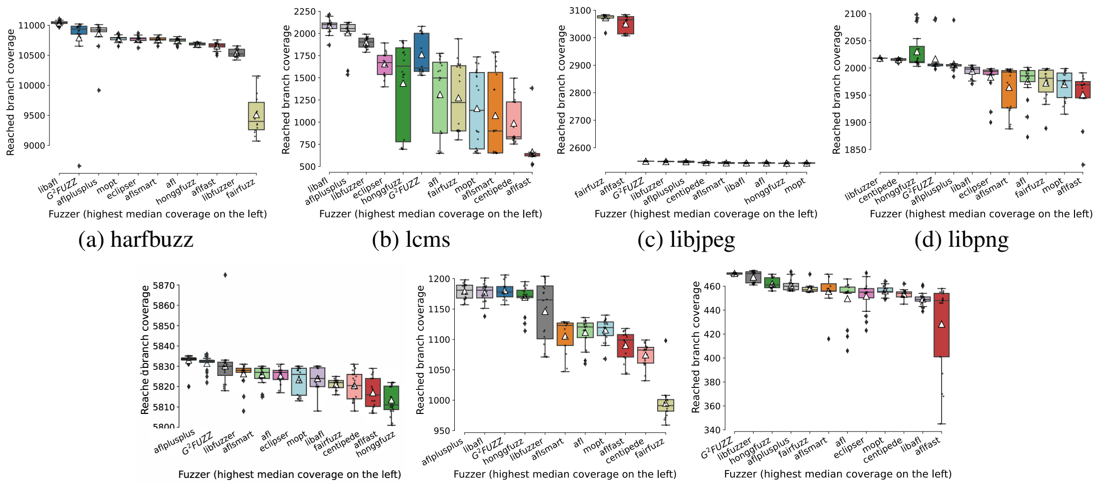
E Library Influence

To evaluate the impact of different libraries on the quality of generator, we conduct experiments across four target formats. Specifically, we use GPT-4 to construct generators, specifying the library to be used in the prompt, such as "*You must use cv2 to create this Python generator.*" For each format, we select two libraries capable of generating files in the corresponding format.

The results are presented in Table 16. In most cases, different libraries exhibit large variations in feature coverage, as observed in the cases of JPG, MP4, and PDF. Notably, combining multiple libraries leads to higher overall feature coverage because their complementary functionalities enable the construction of more sophisticated generators.

Table 16: Feature coverage achieved by using different libraries.

Format	Library	Feature Cov	ValidNum/InvalidNum
JPG	PIL	252	40/0
	cv2	337	40/0
	Unlimited	**984**	130/0
MP4	cv2	**471**	23/11
	moviepy	-	-
	Unlimited	290	19/15
PDF	fpdf	353	27/0
	PyPDF2	72	18/4
	Unlimited	**559**	50/1
TIFF	PIL	164	27/0
	tifffile	161	22/2
	Unlimited	**387**	48/8



(a) harfbuzz (b) lcms (c) libjpeg (d) libpng

(e) openssl (f) woff2 (g) zlib

Figure 12: Code coverage distributions achieved in a FuzzBench experiment.

````
```
<TARGET_GENERATOR>
```
Based on the above code, provide me with a more complex code
that can generate <FROMAT> files with additional more
complex file <features/structures>.
````

Figure 13: The prompt for random mutation.

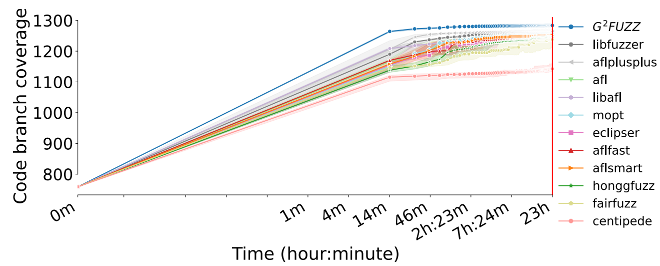


Figure 15: Average Code Coverage Evolution Over Time for *vorbis_decode_fuzzer*.

Table 17: Benchmark programs selected from UNIFUZZ, FuzzBench, and MAGMA.

UNIFUZZ	FuzzBench	MAGMA
gdk-pixbuf-pixdata	bloaty_fuzz_target	libpng_read_fuzzer
jhead	freetype2-2017	read_rgba_fuzzer
mp3gain	harfbuzz-1.3.2	tiffcp
ffmpeg	lcms-2017-03-21	pdf_fuzzer
tiffsplit	libjpeg-turbo-07-2017	pdfimages
pdftotext	libpcap_fuzz_both	pdftoppm
mp42aac	libpng-1.2.56	sndfile_fuzzer
flvmeta	openssl_x509	
imginfo	vorbis-2017-12-11	
exiv2	woff2-2016-05-06	
	zlib_zlib_uncompress_fuzzer	

Table 18: The additional analysis time of LLMGenFuzz. The unit is seconds.

Programs	Group I	Group II	Group III	Average(ExtraPCT[1])
lcms	131	299	158	196(0.22%)
woff2	431	434	220	361(0.42%)
vorbis	552	329	194	358(0.41%)
freetype2	817	788	410	671(0.77%)
libpcap	261	173	220	218(0.25%)
bloaty	743	990	1,043	925(1.06%)
harfbuzz	465	651	282	466(0.54%)
libjpeg-turbo	575	163	118	285(0.33%)
libpng	379	158	47	194(0.22%)
openssl	1,259	189	551	666(0.77%)
zlib	220	161	110	163(0.19%)

[1] **ExtraPCT**: The percentage of the additional analysis time compared to the 23 hours of fuzzing time.

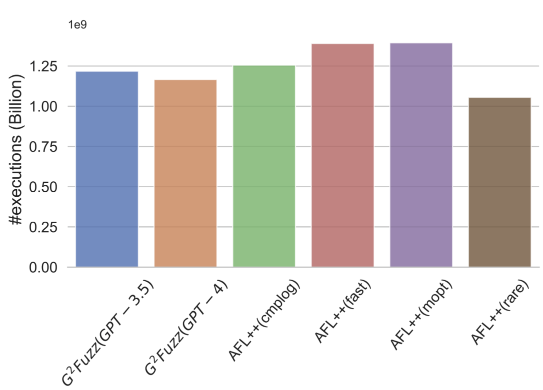


Figure 14: The total throughput of different fuzzers running for 24 hours.

Table 19: The median code coverage achieved by G^{2}FUZZ at 23 hours and 23 hours and 45 mins.

Programs	23 hours	23 hours and 45 minutes	Diff
bloaty_fuzz_target	6,377.0	6,377.0	0
freetype2_ftfuzzer	11,630.0	11,630.0	0
harfbuzz_hb-shape-fuzzer	10,935.5	10,935.5	0
lcms_cms_transform_fuzzer	1,610.0	1,610.0	0
libjpeg-turbo_libjpeg_turbo_fuzzer	2,551.0	2,551.0	0
libpcap_fuzz_both	3,003.0	3,003.0	0
libpng_libpng_read_fuzzer	2,006.0	2,006.0	0
openssl_x509	5,833.0	5,833.0	0
vorbis_decode_fuzzer	1,283.5	1,283.5	0
woff2_convert_woff2ttf_fuzzer	1,178.0	1,178.0	0
zlib_zlib_uncompress_fuzzer	471.0	471.0	0