LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing

Hongxiang Zhang and **Yuyang Rong** and **Yifeng He** and **Hao Chen**
University of California, Davis
Davis, California, USA
{hxxzhang, PtrRong, yfhe, chen}@ucdavis.edu

Abstract

Greybox fuzzing has achieved success in revealing bugs and vulnerabilities in programs. However, bit-level randomized mutation strategies have limited the fuzzer's performance on structured data. Specialized fuzzers can handle specific structured data, but require additional efforts in grammar and suffer from low throughput. In this paper, we explore the potential of utilizing Large Language Models (LLMs) to enhance greybox fuzzing for structured data. We utilize the pre-trained knowledge of LLM about data conversion and format to generate new valid inputs. We further enhance the LLM on structured formats and mutation strategies by fine-tuning with paired mutation seeds. Our LLM-enhanced fuzzer, LLAMAFUZZ, integrates the power of LLM to understand and mutate structured data to fuzzing. Our experiments show that LLAMAFUZZ outperformed the state-of-the-art methods on all benchmarks, demonstrating its effectiveness in various scenarios.

1 Introduction

Fuzz testing, also known as fuzzing, is an automated software testing technique that generates test seeds to discover vulnerabilities in the target software applications. In recent years, greybox fuzzing has drawn much attention because of its effectiveness in discovering new vulnerabilities in many programs. As software systems continue to grow in complexity and evolve at an accelerated pace, the need for adapted test inputs has become increasingly important. While bit-level randomized mutation (Zalewski, 2016; Fioraldi et al., 2020b; Swiecki; Lyu et al., 2019; Lemieux and Sen, 2018) has achieved a lot, they reached a bottleneck in which traditional greybox fuzzers struggle to effectively generate structured data.

General-purpose greybox fuzzers employ high-throughput bit-level mutation. AFL++ (Fioraldi et al., 2020b), one of the state-of-the-art greybox fuzzers, combines multiple mutation strategies and scheduling strategies, leading fuzzing to a new level. However, when fuzzing applications that require structured input, blind random bit-level mutation can be problematic. Such mutations often disrupt the integrity of data formats, resulting in inefficient exploration of the input space. As a result, converging to a high and stable coverage and reaching bugs is time-consuming. To accelerate this process, honggfuzz (Swiecki) shares the file corpus across multi-process and multi-thread execution to boost throughput. However, the effectiveness of merely increasing throughput and adding new random mutations is limited, since the bottleneck is caused by the complex constraints when handling structured seeds. AFL++ and honggfuzz require excessive attempts to generate valid structured inputs.

Therefore, structural awareness of the test seeds is the key to success in fuzzing such software. To generate structured binary seeds, specialized fuzzers use predefined grammars. Gramatron (Srivastava and Payer, 2021) restructures the grammar to enable unbiased sampling from the input state space and permits more aggressive mutation operations. Moreover, it combines search-based testing to co-evolve the test case generation. However, it requires additional specifications in predefined Chomsky Normal Form (Chomsky, 1959) and Greibach Normal Form (Greibach, 1965) to construct grammar automata. NAUTILUS (Aschermann et al., 2019) employs grammar-aware mutation operators to probe deep program paths and reveal complex bugs, while GRIMOIRE (Srivastava and Payer, 2021) synthesizes input structures to discover bugs in grammar-based formats.

Fuzzer developers face a trade-off between employing general-purpose fuzzers and specialized ones. General-purpose fuzzers, while versatile, often struggle with handling structured seeds effectively. Meanwhile, specialized fuzzers can pro-

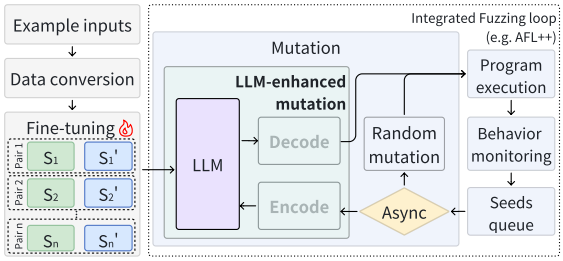

Figure 1: The overview of fuzzing with LLAMAFUZZ. Each fine-tuning sample is a pair (S_i, S_i'), representing a seed before and after successful mutation. LLAMAFUZZ applies dual-layered, asynchronous mutations—traditional fuzzing and LLM-enhanced. The four light-blue boxes on the right (Execution, Behavior Monitoring, Mutation, Seeds Queue) depict the fuzzing loop.

duce high-quality structured seeds, but this specialization can limit their flexibility and applicability. Moreover, relying on grammar rules for seed generation requires extensive domain knowledge, which can be a barrier to their widespread use.

To address the aforementioned challenges, we propose an LLM-enhanced mutation strategy applicable to both binary and text-based formats with minimal fine-tuning. Figure 1 provides an overview of LLAMAFUZZ architecture. Pre-trained on diverse datasets, LLMs can learn intricate patterns for data conversion and data format, which are crucial for structured data mutation. We further fine-tune LLMs to learn specific seed patterns and mutate structured seeds, aiming for a balance between generic and specialized fuzzers. As illustrated in Table 1, random mutations often perform low-level bit flips that corrupt the syntactic structure of valid inputs. In contrast, LLAMAFUZZ produces structurally valid and semantically meaningful variants, enabling more effective and targeted fuzzing.

We integrate LLMs and fuzzer with a lightweight asynchronous job queue, allowing LLAMAFUZZ to run efficiently on single or multiple GPUs. To assess bug-finding capabilities, we compared LLAMAFUZZ against state-of-the-art fuzzers on bug-based benchmark Magma (Hazimeh et al., 2020), including AFL++, Moptafl, Honggfuzz, and Fairfuzz. LLAMAFUZZ outperformed its top competitors, discovering 41 bugs on average and 47 unique bugs overall. In addition, we evaluated LLAMAFUZZ on real-world programs across various structured data formats to access its versatility (Metzman et al., 2021). LLAMAFUZZ performs competitively with grammar-based fuzzers and outperforms AFL++ on 11 out of 15 fuzzing targets.

Table 1: Mutation example on XML. Format corruptions are highlighted in red, while successful and structurally valid mutations are highlighted in green.

Sample Input	Random Mutation	LLAMAFUZZ Mutation
<doc> <message> <to>ace</to> <from>tom</from> <head>Reminder </head> <body>This is a reminder!</body> </message> </doc>	<doc> <message> <to>ace<to> <from>tom</from> <head>Reminder </head> <body>A rem!nder! </bod> </messag> </doc>	<?xml version="1.0" encoding ="UTF-8"?> <doc> <message> <to>alex</to> <from>ben</from> <head>Hello</head> <body>This is a hello! </body> </message> <message> <to>ben</to> <from>alex</from> <head>Reply: Hello </head> </message> </doc>

2 Background and related work

Greybox coverage-guided fuzzing tools (Zalewski, 2016; Fioraldi et al., 2020b; Xu et al., 2024; Pham et al., 2019; Swiecki) employ mutation-based strategies to explore input spaces; however, their reliance on random bit-level mutations often leads to inefficiencies and difficulties in generating valid inputs. To mitigate this, these tools use bitmaps to record execution paths and guide mutations toward unexplored code (Wang et al., 2019a), although traditional bit-level approaches still struggle with highly structured data. Grammar-based fuzzing (Srivastava and Payer, 2021; Blazytko et al., 2019; Fioraldi et al., 2020a; Aschermann et al., 2019) overcomes these challenges by using human-specified grammars and operators to generate syntactically valid and diverse inputs.

Recent advances with LLMs offer a promising alternative by effectively capturing complex data structures to guide seed mutations (Deng et al., 2023; Xia et al., 2024, 2023; Huang et al., 2024; Yang et al., 2024b), as demonstrated by CHATFUZZ (Hu et al., 2023) and compressed-language models (Pérez et al., 2024), thereby enhancing both efficiency and bug discovery in fuzzing. Our method leverages LLMs to learn valid input structures, enabling dynamic, structure-preserving mutations. This approach boosts mutation efficiency, code coverage, and bug detection compared to greybox and grammar-based techniques.

3 Method

3.1 Architecture

We introduce LLAMAFUZZ, an LLM-enhanced greybox fuzzer designed to mutate structured data efficiently. As illustrated in Figure 1, our approach consisted of two primary stages. First, paired struc-

tured data was pre-processed (Figure 2) and used to fine-tune the LLM, enabling LLMs to learn the underlying structural patterns and mutation transformation. Second, we integrated the fuzzer with LLM, which generated structured seeds based on existing inputs.

Our workflow included three parts: **1. Fine-tune preparation:** We collected training data from diverse sources. Also, we introduced a data conversion method that enables the LLM to mutate various data formats. **2. Fine-tuning LLM for mutation:** We fine-tuned the LLM to perform structure-aware mutations. **3. Integrate fuzzer and LLM:** We integrated the fuzzer with the LLM via an asynchronous communication mechanism.

Notably, we excluded seeds used in evaluation from the fine-tuning datasets to preempt potential data leakage, limiting the possibility that the LLM replays memorized seeds to trigger bugs.

3.2 Fine-tune preparation

We followed the LLMs training process by generative pre-training of a language model on a diverse corpus of unlabeled text, followed by discriminative fine-tuning on specific tasks (LeCun et al., 2015). The fine-tuning data were collected from real-world fuzzing processes (Metzman et al., 2021). We used these data to teach LLM the pattern and mutation of structured data, enabling LLM to modify a given seed to generate valuable seeds while keeping the original structure.

Fine-tuning data collection We expected the LLM to be able to understand the structure of data and generate structured seeds for testing, thus we needed to collect a training set first. Specifically, we collected valuable seeds from FuzzBench (Metzman et al., 2021) history experiment data and AFL++ fuzzing data that (1) found new paths, (2) had different hit-counts, (3) triggered crashes. The reasons are intuitive: improving coverage will help fuzzer explore target programs to find vulnerabilities in the unvisited path since bugs can not be found in undiscovered paths. While seeds with different hit counts[1] may not directly improve coverage, they execute the program in varied ways, potentially uncovering vulnerabilities in already visited paths. Ultimately, the goal of fuzzing is to find vulnerabilities, so any seed that triggers crashes is valuable.

[1]number of times a seed exercises a specific path

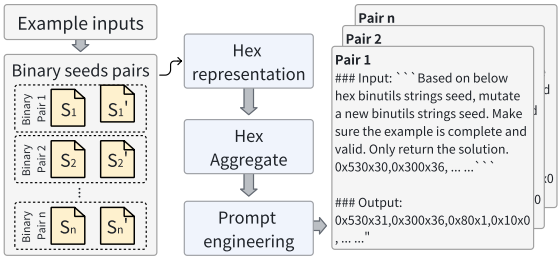

Figure 2: The workflow of dataset pre-processing. Each binary seed pair consists of the original seed S_1 and its mutated version $S_1{}'$. Pairs 1 to n on the right represent entries in the fine-tuning dataset.

Data conversion and pre-processing We construct a generic seed mutation model by converting binary input files to a uniform hexadecimal representation following (Pérez et al., 2024). This conversion serves three purposes. First, to enable LLAMAFUZZ to handle various data formats, we need a uniform data reading method, as adapting to each format is impractical. Second, traditional fuzzers operate at the bit level on binary seeds, whereas LLMs typically require natural language input. Therefore, it is essential to convert the training data to a format that LLMs can understand. Third, the conversion is expected to be efficient, as delay would directly impact fuzzing throughput. Compared to other encoding schemes like base64, hexadecimal is more intuitive, easier to convert from binary. Note that this conversion applies only to binary data; for text-based data, we only add prompts to the seeds, as CHATFUZZ (Hu et al., 2023) has shown LLM can process them directly.

As shown in Figure 2, our data conversion process involves three steps. Initially, the binary seed file is converted into a hexadecimal representation. Next, every two contiguous hexadecimal characters are combined into a single unit. This approach not only reduced the token length of the input string, which was crucial given the limited maximum input token length of most current LLMs but also minimized the need to add new vocabulary to the tokenizer, compared to combining three or more hex characters. Finally, we add a prompt to each fine-tuning instance. In addition to data conversion, we incorporate noise data into our training set to mitigate overfitting and training data replay. Each training example consists of a pair of seeds: the original seed and its corresponding mutated version. This setup enables the LLM to learn both the mutation transformation and the underlying structure of the data formats.

Tokenization As aforementioned, each pair of contiguous hexadecimal characters was compiled into a phrase. However, not all such phrases were present in the tokenizer's vocabulary. To tackle this, we employed a tokenizer that uses the Byte-Pair Encoding algorithm (Sennrich et al., 2015; Touvron et al., 2023), which splits unknown phrases into smaller subword units. This allows the LLAMAFUZZ to generate and understand various data formats.

3.3 Fine-tuning LLM for mutation

Fine-tuning pre-trained models is a common paradigm for achieving proficiency in specific downstream tasks (LeCun et al., 2015; von Werra et al., 2020; Ding et al., 2023; Han et al., 2024). This process builds upon the pre-trained model's general understanding and adapts it to specific tasks through supervised learning. Similarly, supervised fine-tuning (SFT) is necessary when employing a general-purpose LLM for structured data mutation.

As shown in Figure 2, Pair 1 to Pair n provides example prompts to guide the model in structured data mutation. Each prompt consists of the original structured data and its desired mutated result in hexadecimal representation. Subsequently, we prompt the format keywords allowing LLM to take the general understanding of the format from pre-trained knowledge to do the mutation. Prompt examples are provided in AppendixA.3

Following prior works (Xia et al., 2024; Yang et al., 2024c), we apply SFT (von Werra et al., 2020) to adapt the LLM. The supervised fine-tuning objective is defined as:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log P\Big(y^{(i)} \mid x^{(i)}; \theta\Big) \quad (1)$$

where $x^{(i)}$ denotes the i-th original seed, $y^{(i)}$ its corresponding target mutated seed, and θ represents the model parameters. The model is trained to generate $y^{(i)}$ conditioned on $x^{(i)}$.

Additionally, we apply model quantization (Polino et al., 2018) with mixed-precision $fp16$ and integrate LoRA (Hu et al., 2021) to speedup training and inference while maintaining accuracy. Training details are provided in AppendixA.1.

3.4 Integrate Fuzzer and LLM

Speed is paramount for greybox fuzzers, which can execute hundreds or even thousands of seeds per second (Yun et al., 2018; Zheng et al., 2019). Any additional processes integrated into the greybox fuzzer, could potentially impair overall throughput and negatively impact fuzzing performance. Particularly, LLM generation is slower and more resource-intensive, primarily requiring substantial GPU resources.

To address this speed mismatch, we designed an asynchronous job[2] queue for communication between the LLM and the fuzzer. The fuzzing process is formulated as follows:

$$\begin{aligned} Q_{\text{LLM}} &= \{(x^{(i)}, t_i) \mid x^{(i)} \in X,\, t_i \in T\}, \\ Q_{\text{AFL}} &= \{(x^{(i)}, e_i) \mid x^{(i)} \in X,\, e_i \in E\}, \\ F(t) &= \begin{cases} F_{\text{LLM}}(x^{(i)}) & \text{if } Q_{\text{LLM}} \neq \emptyset, \\ F_{\text{AFL}}(x^{(i)}) & \text{otherwise} \end{cases} \end{aligned} \quad (2)$$

where $x^{(i)} \in X$ denotes the current seed test input chosen by the fuzzer, sets T and E represent the running times of LLM-generated and AFL-generated mutations. Q_{LLM} and Q_{AFL} are the asynchronous job queues for LLM mutation and AFL++ fuzzing respectively, with t_i and e_i representing their timestamps. The function $F(t)$ defines the dual-layer fuzzing strategy. The seed selection process prioritizes test cases from both queues according to their coverage impact[3]:

$$x^{i+1} = \underset{x \in Q_{\text{LLM}} \cup Q_{\text{AFL}}}{\arg\max} \{\text{coverage}(x)\} \quad (3)$$

This asynchronous process eliminates waiting time, enabling the fuzzer to run at high speed without delays from LLM mutation tasks. This ensures that integrating the LLM enhances the fuzzer's capabilities without compromising efficiency. Additionally, the dual-layer structure allows easy replacement of different LLMs. An ablation study is provided in AppendixB.2.

4 Experiment

4.1 Benchmarks and Evaluation Metrics

Magma V1.2 (Hazimeh et al., 2020) is a ground-truth fuzzing benchmark suite based on real programs with real bugs. Table 5 outlines details of fuzz targets, where four columns indicate benchmark, project, fuzz target, and expected file format.

[2]Details asynchronous queue design are provided in AppendixA.2

[3]The coverage function is an abstraction of behavior monitoring in the fuzzing loop, determining whether the current seed: (1) triggers new execution paths, (2) yields different hit-counts, or (3) results in crashes.

In the Magma experiment, we compared LLAMAFUZZ with AFL++, Moptafl, Honggfuzz, and Fairfuzz. All baseline fuzzers except for AFL++[4] were provided in Magma. Following Magma setup, we use the number of detected bugs and the time to trigger them as key metrics.

OSS-Fuzz We evaluated a set of real-world programs from OSS-Fuzz (Serebryany, 2017). For fairness and consistency in the evaluation process (Klees et al., 2018), we evaluated in a standard benchmark, FuzzBench (Metzman et al., 2021). The specific applications are detailed in Table 5. The chosen benchmark encompassed 12 open-source programs that process different structured data in their latest versions. As suggested by Klees et al. (2018) and Hazimeh et al. (2020), each experiment were repeated ten times, each trial lasted for 24 hours. We evaluated code coverage using established measures (Böhme et al., 2022; Klees et al., 2018; Wei et al., 2022), ensuring consistency by reporting branch coverage via afl-cov. Statistical significance was analyzed using the Mann-Whitney U test (Klees et al., 2018), and the Vargha-Delaney statistic (Olsthoorn et al., 2020) quantified effect size. Further details are provided in AppendixA.5.

4.2 Implementation details

To evaluate the potential of LLMs in addressing the limitations of traditional fuzzing for structured data, we implemented LLAMAFUZZ by extending AFL++. We used llama2-7b-chat-hf (Touvron et al., 2023) as the base model, which was powerful and efficient. Since LLAMAFUZZ was built on top of AFL++, any observed difference can be attributed to our changes to the implementation of LLM mutation. To prevent seed memorization, we excluded all experimental seeds from the fine-tuning dataset.

We followed the standard instructions provided by the benchmark developers to build the fuzzing targets (Serebryany, 2017; Metzman et al., 2021; Hazimeh et al., 2020). For real-world programs tested under OSS-Fuzz, we utilized the default initial seed corpus as outlined by OSS-FUZZ (Serebryany, 2017). Similarly, during our experiments with the Magma benchmark, we selected default initial seeds specified by the benchmark developers. More details, source code, and related artifacts are provided in AppendixA.4.

[4]We used a more recent version of AFL++ (version 61e27c6) than the one provided in Magma, ensuring that we have access to the latest enhancements.

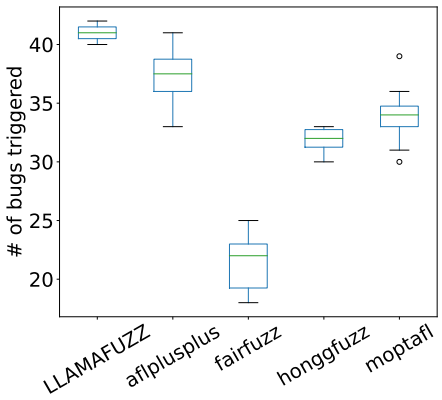

Figure 3: Distribution of the average number of bugs triggered over 24 hours.

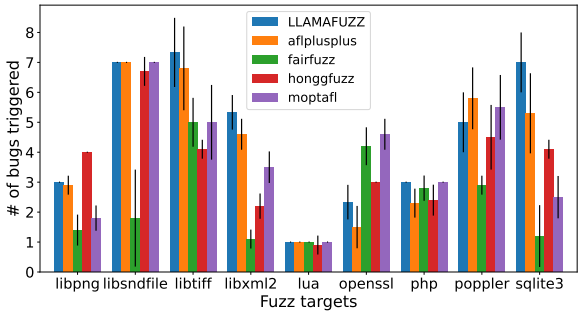

Figure 4: Arithmetic mean number of bugs identified for each project per trial. The black line denotes the 95% confidence interval.

4.3 LLAMAFUZZ find more bugs

We evaluated LLAMAFUZZ performance against other popular fuzzers by checking whether it is able to discover more bugs on Magma. According to Figure 3, LLAMAFUZZ outperforms all other fuzzers in terms of the average number of bugs triggered for each trail. As aforementioned, LLAMAFUZZ was built upon AFL++. Any improvement of LLAMAFUZZ over AFL++ can be attributed to the contribution of LLM. These results highlighted LLAMAFUZZ's competitiveness and robustness with the SOTA in bug-triggering capabilities.

To further investigate the performance of LLAMAFUZZ across different fuzzing targets, Figure 4 presents the arithmetic mean number of bugs identified for each project per trial per day. According to the results, LLAMAFUZZ triggered the most unique bugs among the evaluated fuzzers. It discovered 47 unique bugs in Magma, while AFL++, Moptafl, Honggfuzz, and Fairfuzz found 46, 42, 37, and 31 errors, respectively. Vulnerabilities were found in 9 tested implementations and encompassed various types of memory vulnerabilities, including use-after-free, buffer overflow, and memory leaks. Notably, SQL003, XML006, and

Table 2: Average branch coverage achieved by our approach (LLAMAFUZZ) and the baseline (AFL++). We report the average branch coverage, p-values produced by the Mann-Whitney U Test, and the Vargha-Delaney statistics ($\hat{A}_{12}$). For assessing effect size, we use the labels -, M, and L to represent negligible, medium, and large effects, respectively.

Fuzz target	Fuzz object	Branch coverage (avg)				
		LLAMAFUZZ	AFL++	Improv.	p-value	$\hat{A}_{12}$
binutils	fuzz_nm	13 969	9017	54.91%	<0.01	L(1)
	fuzz_objcopy	22 118	12 494	77.03%	<0.01	L(1)
	fuzz_readelf	6552	4437	47.67%	<0.01	L(1)
	fuzz_strings	6441	5295	21.64%	<0.01	L(1)
bloaty	fuzz_target	5972	5722	4.37%	<0.01	L(1)
freetype2	freetype2-ftfuzzer	10 521	9978	5.45%	<0.01	L(1)
grok	grk_decompress_fuzzer	3721	2313	60.87%	<0.01	L(1)
kamailio	fuzz_parse_msg	3743	2692	39.06%	0.03	L(0.95)
	fuzz_uri	1392	1391	0.04%	0.72	M(0.55)
libavc	avc_dec_fuzzer	9872	9838	0.35%	0.01	L(1)
	mvc_dec_fuzzer	6463	5933	8.94%	0.87	M(0.55)
	svc_dec_fuzzer	11 212	6403	75.11%	0.08	L(0.87)
libhevc	hevc_dec_fuzzer	15 154	15 122	0.21%	0.22	L(0.77)
openh264	decoder_fuzzer	7394	7396	-0.03%	0.79	M(0.57)
zlib	zlib_uncompress_fuzzer	387	385	0.48%	0.60	-

XML002 were never found by any other fuzzers.

To understand the contributions of the LLM mutation, we conducted a more detailed investigation. Bug XML006 (CVE-2017-9048) demonstrated that randomness-only mutation was insufficient; a comprehensive understanding through structure is necessary. XML006 is a stack-based buffer overflow vulnerability in libxml2. To trigger it, the mutator must recursively dump the element content definition into a char buffer *buf* of size *size*. At the end of the routine, the mutator appended two more characters to exceed the *size*. In our experiment, only LLAMAFUZZ triggered this bug, demonstrating that LLAMAFUZZ can facilitate the host fuzzers with the capability of finding bugs.

4.4 Performance comparison on OSS-Fuzz

While performing well on Magma, we are committed to further validating its efficacy in real-world applications. To this end, we have selected a series of open-source programs to conduct a comprehensive evaluation. This step is crucial for demonstrating the practical effectiveness of LLAMAFUZZ's methodologies and techniques in various file formats under real-world conditions.

First, we evaluated the performance of LLAMAFUZZ against specialized grammar-based fuzzers, including Gramatron (Srivastava and Payer, 2021), Grimoire (Blazytko et al., 2019), and Nautilus (Aschermann et al., 2019). We selected a common set of fuzzing targets compatible with all specialized grammar-based fuzzers, including mruby, PHP, and quickjs. Next, we conducted a more comprehensive evaluation using real-world programs. Given that grammar-based fuzzers are limited to well-defined, specialized targets, AFL++ was chosen as the reference competitor, representing the state-of-the-art in greybox fuzzing subsection 4.3.

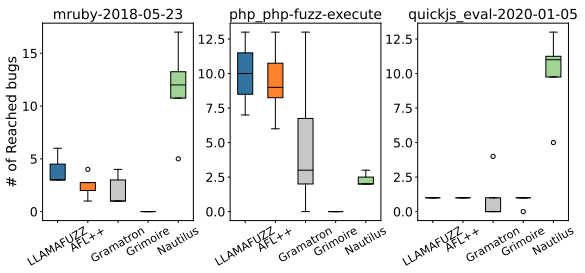

Figure 5: The distribution of reached bug achieved by LLAMAFUZZ, AFL++, Gramatron, Grimoire, and Nautilus.

Compare with grammar-based fuzzers As illustrated in Figure 5, across all three targets, LLAMAFUZZ outperformed the baseline AFL++. Specifically, LLAMAFUZZ identified 13 bugs in PHP, ranking #1 among all evaluated fuzzers. In the case of mruby, LLAMAFUZZ achieved #2 rank. Notably, each selected grammar-based fuzzer demonstrated significant performance only on a subset of the fuzzing targets. For example, Nautilus excelled in mruby and quickjs but did not perform competitively on PHP. In contrast, LLAMAFUZZ showed a strong performance across all targets.

Table 3: Average branch coverage achieved by LLAMAFUZZ, baseline with default initial seeds (AFL++) and baseline with default initial seeds combined with trimmed seeds set from LLAMAFUZZ's fine-tuning dataset (*AFL++**).

Fuzz target	Fuzz object	Branch coverage (avg)					
		LLAMAFUZZ	AFL++	*AFL++**	Improv.	p-value	$\hat{A}_{12}$
binutils	fuzz_nm	13 969	9017	10 983	27.18%	<0.01	L(1)
	fuzz_objcopy	22 118	12 494	13 575	62.93%	<0.01	L(1)
	fuzz_readelf	6552	4437	5789	13.79%	<0.01	L(1)
	fuzz_strings	6441	5295	5220	23.39%	<0.01	L(1)
bloaty	fuzz_target	5972	5722	5918	0.90%	0.37	M(0.7)
freetype2	freetype2-ftfuzzer	10 521	9978	10 838	-2.92%	0.01	-
grok	grk_decompress_fuzzer	3721	2313	2629	41.54%	0.02	L(1)
kamailio	fuzz_parse_msg	3743	2692	2521	48.49%	<0.01	L(1)
libavc	avc_dec_fuzzer	9872	9838	9847	0.26%	0.01	L(1)

Compare with AFL++ To conduct a more comprehensive evaluation of LLAMAFUZZ on real-world programs, we selected AFL++ as the reference competitor, representing the state-of-the-art in greybox fuzzing.

Table 2 reports the average branch coverage, the percentage improvement in average branch coverage over the same timeframe (see column **Improv**), p-value, and $\hat{A}_{12}$. In 11 out of 15 targets, LLAMAFUZZ shows statistically significant improvement compared with AFL++ in terms of code coverage. Furthermore, across all evaluated components, except for zlib, LLAMAFUZZ exhibited medium to large effect sizes. These results underscore LLAMAFUZZ's effectiveness in enhancing coverage across a variety of applications.

While performance on a few targets, such as zlib, kamailio-uri, and openh264, was comparable to AFL++, improvements were more modest or statistically less significant. These cases may reflect challenges in processing larger or more complex seeds, which exceed the LLM's context capacity in the fine-tuning data for domain-specific formats. We believe these results highlight opportunities to further enhance LLAMAFUZZ through better domain adaptation and LLM scaling.

5 Analysis

We have demonstrated the superiority of LLAMAFUZZ among standard benchmark and real-world programs. Moving forward, we first establish that the performance improvements of LLAMAFUZZ are attributable to LLM's inference capabilities rather than the replay of fine-tuning data. Lastly, we explore how LLM enhances the fuzzing.

5.1 Did the LLM improve fuzzing by memorizing the fine-tuning data?

Although subsection 4.3 and subsection 4.4 demonstrate the effectiveness of LLAMAFUZZ, we need to investigate whether the performance gains are due to fine-tuning data replay. To address this, we conducted an ablation study focusing on the significant performance improvements observed in subsection 4.4. We used default initial seeds from OSS-FUZZ used by LLAMAFUZZ and compared it to the default initial seeds combined with trimmed seeds set from LLAMAFUZZ's fine-tuning dataset as initial seeds for AFL++ (see column *AFL++** in Table 3). If LLAMAFUZZ were naively replaying fine-tuning data, we would expect *AFL++** to outperform LLAMAFUZZ across all fuzzing targets.

As detailed in Table 3, LLAMAFUZZ significantly outperformed *AFL++** across all targets except for bloaty and freetype2, demonstrating that LLM enhances the fuzzing process by interpreting structured data rather than simply replaying fine-tuning data. In the cases of freetype2 and bloaty, however, the performance was similar. This is attributed to the fact that trimmed initial seeds set allowed *AFL++** to achieve high coverage in the early fuzzing process, which introduces a bias in favor of *AFL++**. Despite this, the comparable final performance demonstrates that LLM augments the fuzzing process through its intrinsic understanding and inference capabilities.

Interestingly, we observed a slight performance decline in kamailio and binutils_fuzz_strings when comparing *AFL++** with AFL++. This may due to the inherent randomness in the AFL++. On the other hand, this also indicates that the fine-tuning dataset collection has successfully avoided overlap with the testing data.

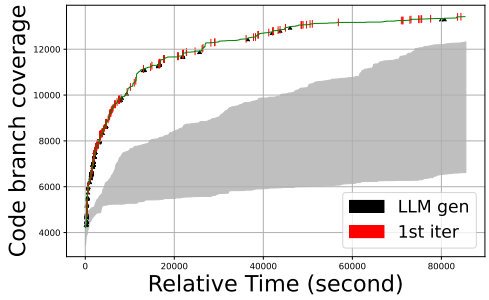

Figure 6: Code coverage growth of LLAMAFUZZ and AFL++ on the binutils-nm target over time. Black triangles mark LLM-generated seeds, while the red vertical line highlights seeds directly sourced from LLM. The green line represents LLAMAFUZZ's coverage, and the grey background indicates AFL++'s coverage range.

5.2 How did LLM augment the fuzzing process?

During the fuzzing process, seeds that can trigger new behavior are considered valuable and are used for further fuzzing. Therefore, understanding the relationship between these seeds and code coverage improvements is crucial for optimizing the fuzzing process. Figure 6 shows the growth of coverage over time. The seeds generated by LLM and the AFL++ seeds sourced from LLM seeds span the whole fuzzing process, which brings steady coverage improvement. This indicates that LLM-generated seeds not only directly impact the fuzzing process but also have a profound and indirect influence on its development. When compared to the grey area, which represents the interval of AFL++ coverage, LLAMAFUZZ achieved both higher and faster coverage. This observation further reinforces the LLM can understand the format structure and mutation strategies during fine-tuning instead of replaying the fine-tuning data. It also aligns with the outcomes from previous experiments conducted in subsection 4.4.

5.3 Case study

To illustrate the advantages of LLMAFUZZ when handling complex data formats, we present an additional example in Table 4 focusing on the highly structured binary format, PNG, and its target program, libpng. PNG files begin with an 8-byte signature and then a series of ordered, CRC-protected chunks (e.g., `IHDR`, `IDAT`, `IEND`). Any effective mutation must preserve format validity while modifying semantic content to expose hidden bugs.

Maintains Structural Validity The PNG format's integrity hinges on its initial 8-byte fixed signature. As demonstrated in the comparison,

Table 4: Byte-level comparison of an original PNG seed and its AFL- and LLAMAFUZZ-mutated variants.

Chunk	Original Bytes	AFL-Mutated Bytes	LLAMAFUZZ-Mutated Bytes
Signature	89 50 4E 47 0D 0A 1A 0A	88 50 4F 47 0D 0A 1A 0A	89 50 4E 47 0D 0A 1A 0A
IHDR	00 00 00 0D 49 48 44 52 00 00 00 20 00	48 49 45 52 01 00 00 21 21 03 00 00	00 00 00 0D 49 48 44 52 00 00 00 5C 00
Image data	00 00 20 02 00 00 00 00 44 A4 8A C6 00 00 00 04 67 41 4D 41 00 00 9C 40 20 0D E4 CB 99 FF CC 3E 4C AF 15 00 63 64 5A 73 6D B6 62 3C 7A F1 A0 73 CD 40 20 B4 0A 0A 0A 0A	01 45 A5 8B C7 00 00 00 04 66 41 4C 41 00 00 9D 41 21 0C E5 CB 98 FF CD 3F 4D AE 14 00 62 64 5B 72 6C B7 63 3D 0B 0B 0B 0B	00 00 44 08 06 00 00 00 58 92 D5 E4 00 00 00 04 67 41 4D 41 00 00 B1 8F 14 83 66 05 54 05 05 05 05 56 DA 2E 30 01 01 95 05 56 DA 2E 30 01 01 95 4D 01 01 6A 98 01 01 08

AFL's bit-level mutation often corrupts this signature (changing '89' to '88'), rendering the file immediately invalid and preventing deeper program logic from being tested. In contrast, LLAMAFUZZ preserves the correct signature, ensuring the mutated input can be processed by the target program.

Semantically-Aware Chunk Mutation PNG files are composed of well-defined chunks with specific ordering and formatting requirements. LLAMAFUZZ selectively targeting important chunks, such as IHDR, and mutating their semantic content (altering the filter method from '20' to '5c') without disrupting the essential chunk boundaries. Conversely, AFL's random bit-level mutations frequently corrupt chunk lengths, types, or CRC checksums, leading to invalid inputs.

By preserving structural correctness and injecting semantically meaningful mutations, LLAMAFUZZ generates inputs that explore the target program's codebase more deeply. This increases the probability of uncovering vulnerabilities that random mutation strategies would likely miss.

6 Conclusion

Mutation is a critical step in greybox fuzzing that directly impacts performance. Although randomized bit-level mutations work well in many cases, state-of-the-art fuzzers struggle with structured data because generating valid, highly structured inputs typically requires many attempts and relies heavily on randomness. In this paper, we propose leveraging LLMs to learn structured data patterns and guide seed mutation. We evaluate LLAMAFUZZ on a ground-truth fuzzing benchmark and a diverse set of real-world programs handling structured data. Our method achieves significantly higher coverage and identifies 47 unique bugs across all trials. These findings confirm LLAMAFUZZ's effectiveness in structure-aware mutation.

7 Limitation

As aforementioned, our dual-layer structure allows easy replacement of different LLMs, including both open-source and closed-source models. We present an ablation study in AppendixB.2 to evaluate the generalizability and effectiveness of our approach across different LLMs, including llama2-7b-chat-hf (Touvron et al., 2023), LLaMA-3-8B-Instruct(Dubey et al., 2024), Mistral-7B-Instruct-v0.2(Jiang et al., 2023), and Qwen2-7B-Instruct(Yang et al., 2024a). A full re-run of every experiment on every model is left to future work as resources permit, but our representative trials already demonstrate that the proposed method consistently transfers across architectures.

References

Andrea Arcuri, Man Zhang, and Juan Galeotti. 2024. Advanced white-box heuristics for search-based fuzzing of rest apis. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–36.

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

Daniel Blackwell, Ingolf Becker, and David Clark. 2025. Hyperfuzzing: black-box security hypertesting with a grey-box fuzzer. *Empirical Software Engineering*, 30(1):1–28.

Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344.

Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1621–1633.

Anne Borcherding, Martin Morawetz, and Steffen Pfrang. 2023. Smarter evolution: Enhancing evolutionary black box fuzzing with adaptive models. *Sensors*, 23(18):7864.

Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors. *arXiv preprint arXiv:2402.03704*.

Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 183–198.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*.

Noam Chomsky. 1959. On certain formal properties of grammars. *Information and control*, 2(2):137–167.

Addison Crump, Andrea Fioraldi, Dominik Maier, and Dongjia Zhang. 2023. Libafl libfuzzer: Libfuzzer on top of libafl. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 70–72. IEEE.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5(3):220–235.

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

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation. *ArXiv*, abs/2402.12222.

Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 337–350.

Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020a. Weizz: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 1–13.

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020b. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Sheila A Greibach. 1965. A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM (JACM)*, 12(1):42–52.

Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. Quickfuzz: An automatic random fuzzer for common file formats. *ACM SIGPLAN Notices*, 51(12):13–20.

Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608*.

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3).

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458.

Allen D Householder and Jonathan M Foote. 2012. Probability-based parameter selection for black-box fuzz testing. *Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2012-TN-019*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782*.

Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE.

Linghan Huang, Peizhou Zhao, Huaming Chen, and Lei Ma. 2024. Large language models based fuzzing techniques: A survey. *arXiv preprint arXiv:2402.00350*.

Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. Tiff: using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 505–517.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed grey-box fuzzing guided by data dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4931–4948.

George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138.

Koffi Anderson Koffi, Vyron Kampourakis, Jia Song, Constantinos Kolias, and Robert C Ivans. 2024. Structuredfuzzer: Fuzzing structured text-based control logic applications. *Electronics*, 13(13):2475.

Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2021. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes*, 44(4):14–14.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature*, 521(7553):436–444.

Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 475–485.

Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity*, 1:1–13.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Hangtian Liu, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Xiangzhi Wang, and Guangming Gao. 2024. Labrador: Response guided directed fuzzing for black-box iot devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 127–127. IEEE Computer Society.

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for c and c++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25.

Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. 2018. Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606*.

Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966.

Ali Madani, Ben Krause, Eric R Greene, Subu Subramanian, Benjamin P Mohr, James M Holton, Jose Luis Olmos, Caiming Xiong, Zachary Z Sun, Richard Socher, et al. 2023. Large language models generate functional protein sequences across diverse families. *Nature Biotechnology*, 41(8):1099–1106.

Sanoop Mallissery and Yu-Sung Wu. 2023. Demystify the fuzzing methods: A comprehensive survey. *ACM Computing Surveys*, 56(3):1–38.

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

Sachit Menon and Carl Vondrick. 2022. Visual classification via description from large language models. *arXiv preprint arXiv:2210.07183*.

Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, New York, NY, USA. Association for Computing Machinery.

Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1224–1228.

Juan C Pérez, Alejandro Pardo, Mattia Soldan, Hani Itani, Juan Leon-Alcazar, and Bernard Ghanem. 2024. Compressed-language models for understanding compressed file formats: a jpeg exploration. *arXiv preprint arXiv:2405.17146*.

Andrea Pferscher and Bernhard K Aichernig. 2022. Stateful black-box fuzzing of bluetooth devices using automata learning. In *NASA Formal Methods Symposium*, pages 373–392. Springer.

Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997.

Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Kostya Serebryany. 2017. {OSS-Fuzz}-google's continuous fuzzing service for open source software.

Ji Shi, Zhun Wang, Zhiyao Feng, Yang Lan, Shisong Qin, Wei You, Wei Zou, Mathias Payer, and Chao Zhang. 2023. {AIFORE}: Smart fuzzing based on automatic input format reverse engineering. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4967–4984.

Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108*.

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

Robert Swiecki. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, and Shengyi Huang. 2020. Trl: Transformer reinforcement learning. `https://github.com/huggingface/trl`.

Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019a. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019b. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*, pages 995–1007.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *CoRR*.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. *Proc. IEEE/ACM ICSE*.

Hang Xu, Liheng Chen, Shuitao Gan, Chao Zhang, Zheming Li, Jiangan Ji, Baojian Chen, and Fan Hu. 2024. Graphuzz: Data-driven seed scheduling for coverage-guided greybox fuzzing. *ACM Transactions on Software Engineering and Methodology*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. White-box compiler fuzzing empowered by large language models. *arXiv preprint arXiv:2310.15991*.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024b. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):709–735.

Liqun Yang, Jian Yang, Chaoren Wei, Guanglin Niu, Ge Zhang, Yunli Wang, Linzheng ChaI, Wanxu Xia, Hongcheng Guo, Shun Zhang, et al. 2024c. Fuzzcoder: Byte-level fuzzing test via large language model. *arXiv preprint arXiv:2409.01944*.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761.

M. Zalewski. 2016. American fuzzy lop - whitepaper.

Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-box fuzzing rpc-based apis with evomaster: An industrial case study. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–38.

Xiangwei Zhang, Junjie Wang, Xiaoning Du, and Shuang Liu. 2024. Wasmcfuzz: Structure-aware fuzzing for wasm compilers. In *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, pages 1–5.

Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for jvm testing. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1133–1144.

Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114.

A Implementation details

A.1 Fine-tuning

For fine-tuning, we utilized a dataset of approximately 10,000 samples per target. The base model, LLaMA 2-7B, was fine-tuned using LoRA (Low-Rank Adaptation) with an adaptation rank of 8, lora_alpha set to 16, and a dropout rate of 0.05. The training was conducted over 20 epochs with a batch size of 1 per device and a maximum sequence length of 1400 tokens on a single A100 GPU. Overall, the fine-tuning process took around 6 hours per fuzzing target. We employed a cosine learning rate scheduler with a learning rate of 2e-4, 30 warm-up steps, and a weight decay of 0.001. Optimization was performed using AdamW. Gradient checkpointing was enabled to reduce memory usage, and 4-bit quantization with bfloat16 compute precision was applied for efficiency. To ensure consistency across runs, branch coverage was computed using afl-cov, and the final model checkpoint was saved for deployment.

A.2 Asynchronous Queue Design

As summarized in Algorithm 1, our hybrid fuzzer orchestrates three concurrent components—Main Fuzzing Loop, LLM Mutation Thread, and AFL Mutation Thread. This design decouples resource-intensive, semantics-driven mutations from the high-throughput fuzzing loop, enabling non-blocking operation and easy scaling or replacement of the LLM backend.

Main Fuzzing Loop The Main Fuzzing Loop continuously selects and executes mutated seeds. Whenever the LLM queue (Queue_LLM) contains entries, those seeds take precedence; otherwise, it falls back to the AFL queue (Queue_AFL). Each seed is used to drive the target program, and a behavior monitor evaluates the execution trace. Seeds that trigger new or anomalous behavior are deemed *interesting* and are re-enqueued into both Queue_LLM_unmutated and Queue_AFL_unmutated, ensuring they undergo further rounds of LLM- and AFL-based mutation.

LLM Mutation Thread This thread operates asynchronously on GPU resources. It dequeues seeds marked for LLM-based mutation, converts them into a hexadecimal representation, and utilizes a fine-tuned LLM to generate structured mutations. The resulting mutated seeds are then added back to the main LLM queue. This separation allows for computationally intensive LLM mutations to occur without blocking the faster main fuzzing loop.

AFL Mutation Thread Running asynchronously on CPU resources, this thread handles AFL-based mutations. It dequeues seeds designated for AFL mutation, applies AFL's mutation strategies, and enqueues the resulting seeds into the AFL queue.

By isolating CPU-bound AFL work from GPU-bound LLM tasks, the framework preserves the high throughput of the fuzzing pipeline while benefiting from both mutation paradigms.

A.3 Fine-tuning Prompt example

Input: "'Based on below hex **{fuzzing_target}** seed, mutate a new **{fuzzing_target}** seed. Make sure the example is complete and valid."
{hex_seed}
Output:

A.4 Implementations

As described in subsection 3.4, we developed an asynchronous job queue to communicate between the fuzzer and LLM. Besides, we revised AFL++ source code (e.g. afl-fuzz-bitmap.c) to incorporate LLAMAFUZZ's functionalities, such as message queue and seeds evaluations.

Following previous work (Radford et al., 2019; Wang et al., 2022), we selected a temperature of 1.0 to mutate the structured data for precise and factual responses. In addition, we adopted the model quantization (Polino et al., 2018) into mixed precision floating-point $16fp$ and enabled Lora (Hu et al., 2021) to freeze some of the parameters, increasing training and inference speed without sacrificing too much accuracy.

A.5 Experiment Setup and Metrics

We chose Magma for several reasons. First, Magma involves a wide range of popular programs that process diverse structured data with real-world environments, including 9 libraries and 21 objects. Second, unlike LAVA-M (Dolan-Gavitt et al., 2016), which primarily employs synthetic bugs and magic byte comparisons, Magma offers a diverse range of real vulnerabilities, with a total of 138 bugs spanning integer errors, divide-by-zero

Algorithm 1 Hybrid Fuzzer with LLM- and AFL-based Mutations

```
Initialization: start_fuzzer(); llm_mutate_thread(); afl_mutate_thread()
function START_FUZZER
    while True do
        if Queue_LLM is not empty then
            seed ← dequeue(Queue_LLM)
        else
            seed ← dequeue(Queue_AFL)
        end if
        if run_fuzzer(seed) yields an interesting result then
            enqueue(Queue_LLM_unmutated, seed)
            enqueue(Queue_AFL_unmutated, seed)
        end if
    end while
end function
function LLM_MUTATE_THREAD
    while True do
        if not Queue_LLM_unmutated.empty() then
            seed ← dequeue(Queue_LLM_unmutated)
            hex_seed ← convert_to_hex(seed)
            mutated_seed ← LLM_generate(hex_seed)          ▷ LLM-based mutation on GPU
            enqueue(Queue_LLM, mutated_seed)
        end if
    end while
end function
function AFL_MUTATE_THREAD
    while True do
        if not Queue_AFL_unmutated.empty() then
            seed ← dequeue(Queue_AFL_unmutated)
            mutated_seed ← AFL_generate(seed)              ▷ AFL-based mutation on CPU
            enqueue(Queue_AFL, mutated_seed)
        end if
    end while
end function
```

Table 5: Targets information. The programs tested under the Magma are utilized in their default versions. For programs included in the real-world bench, we specify the exact versions used by listing the Git SHA identifiers behind each project name.

	Project & version	Fuzz Target	File format
Magma V1.2	libpng	libpng_read_fuzzer	PNG
	libsndfile	sndfile_fuzzer	Audio
	libtiff	tiff_read_rgba_fuzzer tiffcp	TIFF
	libxml2	read_memory_fuzzer xmllint	XML
	lua Targets	lua	Lua
	openssl	asn1, asn1parse, bignum, server, client, x509	Binary blobs
	php	json exif unserialize parser	JSON EXIF Serialize object PHP
	poppler	pdf_fuzzer, pdfimages, pdftoppm	PDF
	sqlite3	sqlite3_fuzz	SQL query
Real-world programs	mruby 14c2179	mruby_fuzzer	mruby
	php afa034d	php_exec	php
	quickjs 91459fb	eval	javascript
	binutils 7320840	fuzz_nm, fuzz_objcopy, fuzz_readelf fuzz_strings	ELF String
	bloaty 34f4a66	fuzz_target	ELF, Mach-O, WebAssembly
	freetype2 cd02d35	ftfuzzer	TTF, OTF, WOFF
	grok b9286c2	decompress_fuzzer	JPEG 2000
	kamailio 3f774f3	fuzz_parse_msg fuzz_uri	sip_msg URI
	libavc 828cdb7	avc_dec_fuzzer mvc_dec_fuzzer svc_dec_fuzzer	AVC MVC SVC
	libhevc d0897de	hevc_dec_fuzzer	HEVC
	openh264 1c23887	decoder_fuzzer	H.264/MPEG-4 AVC
	zlib 0f51fb4	uncompress_fuzzer	Zlib compressed

faults, memory overflows, use-after-free, double-free, and null-pointer dereference scenarios. It incorporates real-world bugs from older versions of software updated to their latest releases, ensuring the benchmark's relevance and practical applicability. Third, Magma focuses on bug counts and time-to-bug metrics as more direct surrogates for performance (Hazimeh et al., 2020). In addition, we chose FuzzBench in later real-world experiments, because it employed Docker containers to standardize the testing environment for each fuzzer. This setup guaranteed fairness that all fuzzers operated under identical conditions, thus ensuring comparability of results.

Table 5 summarizes the fuzzing targets used in our experiments, covering both Magma and real-world programs. The Magma benchmark includes 9 projects, comprising 21 fuzz targets across 12 different file formats. The real-world benchmark features 12 projects, with 18 fuzz targets spanning 19 file formats. Our selection followed three criteria. First, the benchmark had to have a diverse structure format. Second, the program needed to handle complex structured data. Third, the program had to be popular and important.

To assess the statistical significance of our results, we applied the Mann-Whitney U Test (Klees et al., 2018). As per the Mann-Whitney U-test, a result is statistically significant if the p-value is less than 0.05. In addition, we used the Vargha-Delaney statistic (Olsthoorn et al., 2020) to quantify effect size, providing insight into the magnitude of observed differences. The result was classified as negligible if $\hat{A}_{12}$ was less than 0.5, as medium if it was greater than 0.5 but not exceeding 0.8, and as large if it was greater than 0.8.

B More Experiment Result

B.1 Performance on bug-triggered time

Consequently, we listed all the unique bugs triggered, including bug ID and the expected time taken to trigger it in Figure 7. The reported time accounts for missed measurements (where the fuzzer only triggered a bug in M out of N campaigns) and fits the distribution of time-to-bug samples onto an exponential distribution (Hazimeh et al., 2020).

Compared to AFL++, LLAMAFUZZ triggered a greater number of bugs and significantly sped up, with darker blue grid cells representing faster bug triggering. Specifically, LLAMAFUZZ achieved significant speedups in 29 of 43 bugs that were

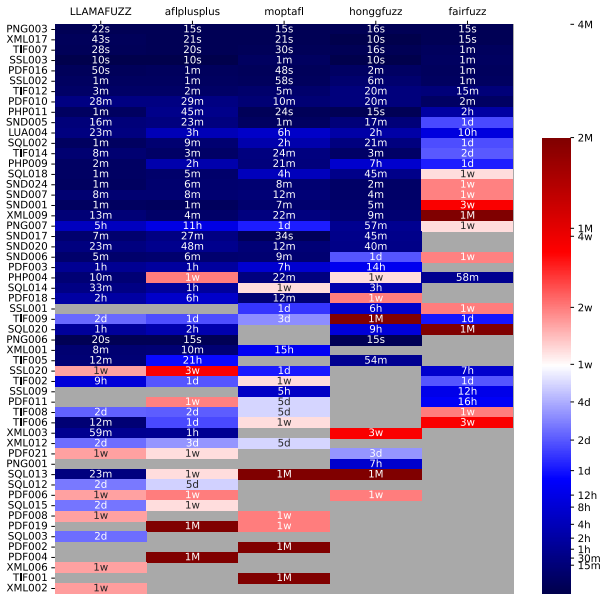

Figure 7: Heatmap of expected bug trigger time achieved by LLAMAFUZZ, AFL++, Moptafl, Honggfuzz, and Fairfuzz at the end of each 24-hour trail. Within each block, more intense blue shades denote shorter expected trigger times, while the grey parts represent bugs that were not triggered in any trials by that fuzzer.

triggered in both LLAMAFUZZ and AFL++ with the remaining bugs exhibiting similar trigger times. In comparison to moptafl, honggfuzz, and fairfuzz, LLAMAFUZZ triggered bugs faster in 25, 23, and 21 cases respectively. Overall, the results indicate a substantial advantage of LLAMAFUZZ over AFL++, Moptafl, Honggfuzz, and Fairfuzz in exploring bugs.

B.2 Different models comparison

Constrained by GPU resources, we selected a small set of real-world programs to demonstrate the generalizability of our method across different LLMs. We select llama2-7b-chat-hf (Touvron et al., 2023), LLaMA-3-8B-Instruct(Dubey et al., 2024), Mistral-7B-Instruct-v0.2(Jiang et al., 2023), and Qwen2-7B-Instruct(Yang et al., 2024a) in this experiment to balance GPU cost. Table 6 outlines the results, comparing the performance of our LLM-enhanced fuzzing approaches against the baseline AFL++. Across all evaluated targets, LLM-enhanced methods outperformed AFL++, achieving higher branch coverage. Notably, more recent LLMs, such as LLaMA3-8B, Mistral, and Qwen-7B, exhibited superior performance, likely due to their improved inference capabilities and better contextual understanding. The result demonstrates the generalizabil-

Table 6: Average branch coverage achieved by LLAMAFUZZ, baseline with default initial seeds (AFL++) and baseline with default initial seeds combined with trimmed seeds set from LLAMAFUZZ's fine-tuning dataset (*AFL++**).

Fuzz target	**Fuzz object**	**Branch coverage (avg)**				
		AFL++	LLAMAFUZZ integrated			
			llama2-7b-chat-hf	LLaMA-3-8B-Instruct	Mistral-7B-Instruct-v0.2	Qwen2-7B-Instruct
binutils	fuzz_nm	9017	13 958	14 423	14 929	15 041
grok	grk_decompress_fuzzer	2313	3750	4028	3839	4123
kamailio	fuzz_parse_msg	2692	3743	3821	3911	4013

ity of our method, confirming its adaptability to various LLMs while consistently enhancing fuzzing effectiveness.

C Related work

C.1 Fuzzing

Fuzzing (Zalewski, 2016; Fioraldi et al., 2020b; Xu et al., 2024; Pham et al., 2019; Swiecki) is an automated random software testing technique to discover vulnerabilities and bugs in the target programs or applications. Traditional fuzzers can be categorized into black-box fuzzers (Feng et al., 2021; Pferscher and Aichernig, 2022; Li et al., 2018; Liu et al., 2024; Borcherding et al., 2023; Householder and Foote, 2012), white-box fuzzers (Yang et al., 2024b; Borkar et al., 2024; Zhang et al., 2023; Arcuri et al., 2024), and grey-box fuzzers (Blackwell et al., 2025; Pham et al., 2019; Böhme et al., 2017; Kim et al., 2023; Fioraldi et al., 2020a; Huang et al., 2022; Wang et al., 2019b) depending on whether fuzzers are aware of the program structure. The black-box fuzzer threat targets a black box, and it's unaware of the program structure. Usually, a black-box fuzzer has a high execution volume since it randomly generates test input, but it only scratches the surface. YARPGen (Livinskii et al., 2020) applies random mutation rigorously applies language specifications to ensure the validity of test cases to test C and C++ compilers. Similarly, Csmith (Yang et al., 2011) generates programs that cover a large subset of C while avoiding undefined and unspecified behaviors.

White-box fuzzers utilize program analysis to improve the code coverage to explore certain code regions, which can be efficient in revealing vulnerabilities in complex logic. WhisperFuzz (Borkar et al., 2024) introduces a static analysis method designed specifically to detect and locate timing vulnerabilities in processors. The tool focuses on evaluating the coverage of microarchitectural timing behaviors, providing a targeted and comprehensive assessment that aids in identifying potential security risks associated with timing flaws.

However, program analysis and defining specialized seed generation grammar could be extremely time-consuming. Greybox fuzzer combines the effectiveness of white-box fuzzer and the efficiency of black-box fuzzer. It leverages instrumentation to get feedback from target programs and leading fuzzers to generate more valuable seeds resulting in higher code coverage. Greybox fuzzers usually combined with mutation strategies rely on iterative modifications of existing seeds to produce novel fuzzing inputs. In addition to basic mutations, recent researchers have developed complex transformations to maintain type consistency (Jain et al., 2018; Chaliasos et al., 2022), adding historical bug-triggering code snippets (Holler et al., 2012; Zhao et al., 2022), and coverage feedback (Aschermann et al., 2019; Fioraldi et al., 2020b) for improved testing efficiency. American Fuzzy Lop (AFL) (Zalewski, 2016) and its variations (Fioraldi et al., 2020b; Lyu et al., 2019; Crump et al., 2023), employ genetic algorithms with a fitness function to prioritize fuzzing inputs for further mutations aimed at enhancing coverage, concentrating on byte-level changes.

C.2 Coverage-guided greybox fuzzing

To overcome the inherent randomness challenges in fuzzing, researchers suggest using bit-map to record coverage information as feedback to more effectively guide the fuzzing process (Zalewski, 2016). Since vulnerabilities cannot be detected in uncovered paths, focusing on expanding the coverage of execution paths is a reasonable step toward improving the performance of fuzzing techniques.

Given a program under test and a set of initial seeds, the coverage-guided greybox fuzzing process mainly consists of four stages.

Seeds queue: a seed was selected from the seeds pool for mutation.

Seed mutation: the selected seed was mutated by various mutation strategies to generate new test seeds.

Execution: Execute the current seed into the program.

Behavior monitoring: Each new seed will be fed into the instrumented program for execution and evaluated by coverage metric. If the seed triggers a new coverage, it will be added to the seeds queue for further fuzzing.

As the fuzzing loop continues, more code branches will be reached, which holds the potential to trigger a bug (Wang et al., 2019a).

C.3 Fuzzing for structured data

Coverage-guided greybox fuzzing has been effective in identifying vulnerabilities in many real-world programs. However, with the increasing complexity of software development, many programs use highly structured data in special formats, which poses significant challenges for traditional fuzzing techniques. Traditional fuzzers primarily perform mutations at the bit level, requiring excessive attempts to mutate such structured data effectively. Moreover, blind random mutation strategies often disrupt the consistency of data formats, leading to the generation of numerous inefficient and ineffective test cases.

Fuzzers for structured data (Wang et al., 2019b; Le et al., 2021; Mallissery and Wu, 2023; Zhang et al., 2024; Meng et al., 2024; Koffi et al., 2024; Shi et al., 2023) can accurately identify the target input format. They can generate test cases that maintain the consistency of the format. This approach ensures that the generated test cases are not only valid but also effective in triggering and exploring potential vulnerabilities or issues within the application. Three grammar-aware mutation operators have been found to be particularly effective in uncovering deep bugs (Srivastava and Payer, 2021; Aschermann et al., 2019): random mutation, which involves selecting a random non-leaf non-terminal node and creating a new context-free grammars derivation subtree. Random recursive unrolling, which finds recursive production rules and expands them up to n times. Splicing, which combines two inputs while preserving their syntactic validity. In addition, Langfuzz (Holler et al., 2012) combines grammar-based fuzz testing and reusing project-specific issue-related fragments, maintaining the integrity of format and having a higher chance to cause new problems than random input. QuickFuzz (Grieco et al., 2016) leverages Haskell's QuickCheck and the Hackage to fuzz structured data. This integration, combined with conventional bit-level mutational fuzzers, negates the need for an external set of input files and eliminates the requirement to develop specific models for the file types being tested. Alternatively, WEIZZ (Fioraldi et al., 2020a) employs a chunk-based mutator to generate and mutate inputs for unknown binary formats. Nevertheless, WEIZZ struggles to handle grammar-based formats such as JSON, XML, and programming languages.

C.4 Augment fuzzing through machine learning

Current research primarily concentrates on two aspects: employing machine-learning models as generators and leveraging machine-learning models to guide the fuzzing process.

C. Pérez (Pérez et al., 2024) explored the ability of Compressed-Language Models (CLMs) to interpret files compressed by standard file formats. Their findings revealed that CLMs are capable of understanding the semantics of compressed data directly from the byte streams, opening a new path for processing raw compressed files. In a related study, CHATFUZZ(Hu et al., 2023) investigates the mutation capabilities of LLM on text-based seeds, achieving 12.77% edge coverage improvement over the SOTA greybox fuzzer (AFL++). Similarly, SmartSeed (Lyu et al., 2018) combines deep learning models to generate new inputs for evaluating 12 different applications.

Prior work (Eom et al., 2024) integrates an LLM-based mutator with a reinforcement learning approach, utilizing the Term Frequency-Inverse Document Frequency technique to develop a weighted coverage map. This method capitalizes on coverage feedback to enhance the effectiveness of the mutation process. Similarly, Xia et al. (Xia et al., 2024) introduce an auto-prompting phase that employs LLMs to produce and mutate test cases across six programming languages. Their findings indicate that LLMs can surpass the coverage achieved by cutting-edge tools.

Additionally, WhiteFox (Yang et al., 2023) employs dual LLMs within their framework: one analyzes low-level optimization source code to inform optimization strategies, while the other generates test programs based on this analysis.

CHATAFL (Meng et al., 2024) utilizes LLMs to understand protocol message types and assesses their ability to identify "states" in stateful protocol implementations. LLM4FUZZ (Shou et al., 2024) leverages LLMs to guide fuzzers towards more critical code areas and input sequences that are more likely to reveal vulnerabilities, showcasing the potential of LLMs in prioritizing and refining fuzzing efforts.

C.5 Large Language Model

In recent studies, pre-trained Large Language Models (LLMs) have shown impressive performance on natural language tasks, including Natural language understanding, reasoning, natural language generation, multilingual, and factuality (Chang et al., 2023; Menon and Vondrick, 2022; Madani et al., 2023; Li et al., 2022).

Utilizing unsupervised learning, Large Language Models are pre-trained on extensive textual data, enabling LLM with a broad range of knowledge. Additionally, with billions or trillions of parameters, LLM can not only capture the patterns in context but also understand the textual data at a deeper level such as format and chunk information within files. Such capabilities have facilitated LLMs to exhibit remarkable competencies beyond traditional Natural Language Processing tasks. Evidence of their versatility includes visual classification (Menon and Vondrick, 2022), protein sequence generation (Madani et al., 2023), code generation (Li et al., 2022).

Building upon this versatile foundation. This inherent capability to interpret and process different data structures renders LLMs particularly effective in the mutation stage of fuzzing processes. CHATFUZZ (Hu et al., 2023) employs LLMs to directly generate seeds, though its application is limited to text-based target programs such as JSON and XML. Moreover, Pérez et al. demonstrate that Compressed-Language Models can understand files compressed by Compressed File Formats.

D Data Availability

D.1 LLAMAFUZZ

The LLAMAFUZZ system comprises two primary components: a greybox fuzzer and a large language model (LLM). The greybox fuzzer (AFL++) in LLAMAFUZZ is available in two versions to cater to different types of seeds:

Binary Seeds Version: This version of the fuzzer is designed to handle binary seeds, providing efficient and effective fuzzing for binary-based targets. https://anonymous.4open.science/r/AFLplusplus-binary-seeds

Text-Based Seeds Version: This version is tailored for text-based seeds, ensuring comprehensive coverage and mutation for text-based targets. https://anonymous.4open.science/r/AFLplusplus-text-based

The link below provided the LLM component of LLAMAFUZZ:

https://anonymous.4open.science/r/LLAMAFUZZ-CDC5

D.2 Experiment benchmarks

We have extended the base version of benchmark frameworks to support LLAMAFUZZ. This includes added functionality to seamlessly integrate the greybox fuzzer and LLM into the benchmarking process.

Magma Experiment Version:
https://anonymous.4open.science/r/magma-llamafuzz/

Fuzzbench Experiment Version:
https://anonymous.4open.science/r/fuzzbench-llamafuzz/