How to Train Your Llama – Efficient Grammar-Based Application Fuzzing Using Large Language Models

Ibrahim Mhiri[1], Matthias Börsig[2][0000−0002−6060−6026], Akim Stark[2][0009−0005−4718−1202], and Ingmar Baumgart[2]

[1] 1&1 Mail & Media, Karlsruhe, Germany ibrahim.mhiri@1und1.de
[2] FZI Research Center for Information Technology, Karlsruhe, Germany {boersig,stark,baumgart}@fzi.de

Abstract. Fuzzing is an automated testing technique that generates random input to identify software bugs and vulnerabilities by provoking unexpected behavior. Although effective, traditional fuzzing lacks input generation guidance, which often leads to inefficiency and wasted time, especially for complex programs, because many inputs are invalid and are rejected. Grammar-based fuzzers address this problem by generating inputs that match the syntactic structure of the program, although they require expert knowledge to define accurate grammars.
Large Language Models (LLMs) show remarkable capabilities in Natural Language Processing (NLP), improving efficiency in various domains. These models can be used to generate input for fuzzers, as they can quickly learn or already have familiarity with the required input formats. This paper explores the integration of LLMs with fuzzing methods to streamline directed input generation and thereby increase fuzzing efficiency. We specifically adapt Llama2 for use with American Fuzzy Lop (AFL), focusing on Extensible Markup Language (XML) due to its commonality as a structured file format. Our approach demonstrates the potential of LLMs to enhance traditional fuzzing by providing targeted, intelligent input generation. Experimental results show that our approach can achieve up to six times more code coverage after 24 hours compared to using AFL alone. Furthermore, in our tests, our method provides up to 50% more coverage than a grammar-based fuzzer.

Keywords: Grammar-Based Fuzzing · XML · Fuzzing · Large Language Models · Llama2 · Fine-Tuning · Prompt-Tuning

1 Introduction

Fuzzing is an important technique in IT security for identifying vulnerabilities in software by automatically testing programs with both expected and malformed inputs. This automated testing approach deliberately inputs a wide range of data to the software, looking for instances where unexpected or faulty responses, such

L. Horn Iwaya et al. (Eds.): NordSec 2024, LNCS 15396, pp. 1–19, 2025.
https://doi.org/10.1007/978-3-031-79007-2_13

as crashes, unhandled exceptions, or security vulnerabilities, are triggered [4]. Grammar-based fuzzing, a specialized testing method, targets applications that handle structured input by using the expected input grammar of the application to generate syntactically correct test cases aimed at exploring the limits of its data handling capabilities.

The recent rise of Large Language Models (LLMs) has significantly impacted various fields by their ability to generate text and to predict sequences of text based on the context provided to them [18]. The application of LLMs to fuzzing, in particular grammar-based fuzzing, offers a new approach to improving software testing by generating complex and varied test inputs that can reveal hidden vulnerabilities beyond the reach of traditional fuzzing methods.

This paper explores the use of LLMs in grammar-based fuzzing to improve vulnerability detection by generating grammar-compliant input. We aim to optimize LLMs for specific tests and refine input generation through feedback, providing a more effective method of testing complex applications than traditional approaches.

We conduct a comparative evaluation against current fuzzing techniques to assess the efficacy of our LLM-enhanced fuzzing approach. The goal is to demonstrate not merely the applicability of LLMs to grammar-based fuzzing but also to highlight the broader implications for testing software that processes complex structured inputs. This investigation underscores the potential of leveraging LLMs to advance the field of software testing, emphasizing the scientific inquiry into their practical application rather than an uncritical endorsement of their capabilities. In summary, our paper makes the following contributions:

- Adaptation of a pre-trained Llama2 13B LLM: As a proof of concept, we adapt a pre-trained Llama2 model to represent the grammar of XML files. This adaptation allows the LLM to generate optimized XML patterns, which are particularly valuable for fuzzing applications that depend on XML data.
- Integration of an LLM into American Fuzzy Lop (AFL): We illustrate how the integration of an LLM for input generation improves the efficiency of the AFL fuzzing framework. This cooperation facilitates the generation of new input files, allowing the fuzzer to explore deeper states more effectively.
- Continuous and dynamic improvement strategy: We propose a feedback loop approach to consistently and dynamically improve the fuzzer's performance throughout the fuzzing process. This approach ensures that the effectiveness of the model is maintained over time.

2 Related Work

Hu et al. [6] introduce ChatFuzz, an extension of grey box fuzzers like AFL++ that integrates generative AI. The system uses large language models to generate XML inputs that conform to the format specifications of programs with structured input. The work uses an LLM (ChatGPT) without further Fine-Tuning or Prompt-Tuning, which performs worse in edge coverage compared to our approach.

The work of Zhang et al. [27] implements LLAMAFUZZ. It also recognizes the ability of LLMs to generate structured input as test cases for fuzzing. However, their approach focuses exclusively on Fine-Tuning LLMs to handle input seed mutations and evaluating performance improvements using AFL++ as the baseline fuzzer. In contrast, our work explores different strategies, such as Prompt-Tuning and Fine-Tuning, to determine which approach yields better results for input generation. Unlike Zhang et al., we do not delegate the mutation process to the LLM; instead, the LLM generates the initial inputs while the fuzzer retains control over the mutations.

Xia et al. [21] present Fuzz4All, a generic fuzzing tool that uses LLM to test different target programs. The tool has two phases: Autoprompting, which refines user input using LLMs, and the fuzzing loop, which generates input for testing. Fuzz4All prioritizes generality over efficiency and is tailored for compilers. This differs from our approach, where we try to learn one grammar at a time and focus on intensive fuzzing.

Meng et al. [10] introduce CHATAFL, an LLM-guided protocol fuzzer using pre-trained models for human-readable specifications. CHATAFL's systematic interaction with LLMs enhances state and code coverage by learning the protocol specification and guiding the fuzzer. Although the Fine-Tuning approach is used to enhance the capabilities of an LLM, it differs from our methodology as it is only a part of our approach. Additionally, it operates in a different domain. While we investigate XML parsers, it focuses on learning a network protocol.

In the field of fuzzing, various methods use LLM to improve specialized fuzzers. Yang et al. [24] introduce KernelGPT, which extends kernel fuzzing by using LLM to automatically infer specifications for Syzkaller, one of the most widely studied kernel fuzzers. Deng et al. [3] present TitanFuzz, a generative LLM that creates seed programs for fuzzing Deep Learning (DL) APIs. TitanFuzz uses a multi-armed bandit algorithm for mutation operations and demonstrates effectiveness in detecting bugs across CPU and GPU backends. In a follow-up paper, Deng et al. [2] introduce FuzzGPT, an extension of TitanFuzz, automating varied input program creation to fuzz DL libraries with LLMs. FuzzGPT improves error detection and has uncovered many bugs in PyTorch and TensorFlow. Liu et al. [9] demonstrated the use of LLMs for fuzz target generation, presenting a generalized approach for input generation.

Le Mieux et al. [7] present CodaMosa, a tool for Search-Based Software Testing (SBST). The paper shows that LLMs can be used to derive new test cases based on existing ones, leading into an increase in the overall test coverage.

Ackerman and Cybenko [1] propose an NLP-driven approach using a LLM to address software vulnerabilities from format specification ambiguity.

Prior to the advent of LLM, significant research and various other efforts, including experiments with Machine Learning (ML) and DL, aimed at improving the efficiency of (grammar-based) fuzzing [4, 16, 19, 20, 23].

3 Background

3.1 Large Language Models

LLMs are trained on large amounts of text data to understand and generate human language. They are based on the transformer model, which uses an encoder-decoder architecture. The encoder processes the input data by converting it into a continuous representation, while the decoder generates the output sequence from this representation. The model uses a self-attention mechanism to process input data in parallel, allowing it to assess the relationships between words, even over long distances in a sentence, and to effectively weigh the importance of words. As a result, LLMs can predict the next word or phrase based on learned patterns and produce coherent output. They excel at tasks such as answering questions, summarizing information and producing structured text.

3.2 Model Tuning

The machine learning community has created open access to a wealth of pre-trained LLMs. These models often encompass a diverse range of both natural (e.g., English, French, German) and programming (e.g., JavaScript, Python) languages, making them versatile tools for various applications. Platforms like Hugging Face[3] offer convenient access to these pre-trained LLMs, which, while operationally ready for immediate tasks, often require further refinement, like Fine-Tuning or Prompt-Tuning, to excel in specific applications.

3.3 Fine-Tuning

Fine-Tuning involves the precise adjustment of the parameters of a pre-trained LLM to tailor it for a particular task or dataset, such as enhancing its capability in sentiment analysis, question-answering, or the generation of structured data like Extensible Markup Language (XML) for fuzzing purposes. This adaptation process necessitates additional training on a target-specific dataset, enabling the model to Fine-Tune its parameters for optimized performance. Although this method is potent, it is also marked by its high demand for resources. The substantial number of parameters in current LLMs, potentially reaching into the billions, entails a significant computational burden. As a result, the process demands specialized computational resources, like advanced Graphical Processing Units (GPUs) or Tensor Processing Units (TPUs), and can extend over prolonged periods, posing potential constraints on its applicability [22].

3.4 Prompt-Tuning

LLMs undergo extensive pre-training across vast datasets, equipping them with a broad exposure to various languages. This pre-training instills in them a foundational capability to process diverse linguistic structures and grammars. In

[3] https://huggingface.co/

many cases, this extensive pre-training makes it inefficient to adjust the entire model for specific applications through Fine-Tuning. Instead, a more targeted approach, known as Prompt-Tuning (a specific Parameter-Efficient Prompt Tuning (PEFT) technique), proves to be more effective [11].

PEFT optimizes a pre-trained LLM for particular tasks by altering only a select subset of the parameters of a model. This process involves introducing specific input-output pairs that guide the model towards generating the desired outcomes for particular tasks. By freezing the original weights of the model and focusing adjustments on a smaller set of prompt parameters, PEFT facilitates the adaptation of the model with minimal modifications. This approach leverages the extensive pre-trained knowledge of the model while ensuring a resource-efficient adaptation process. As models grow in complexity, the advantages of Prompt-Tuning might become even more pronounced, providing a practical method for tailoring LLM to specialized tasks without extensive retraining [8, 26].

3.5 Model Inference

During the inference phase, an LLM uses its trained pattern recognition to generate outputs from given inputs (prompts). This phase demonstrates the model's ability to adapt to new, unseen data. "Inference" here refers to the use of statistical algorithms to produce text based on learned patterns, not to make judgments or inferences [13, 25].

In practical terms, inference is where the theoretical capabilities of an LLM lead to tangible results, such as answering questions, generating text, or creating code. The efficiency and accuracy of inference are critical because they directly affect the real-world utility and effectiveness of the model. Advances in model architectures, training methods, and computational technologies continue to improve LLMs, making them increasingly valuable in a variety of industries.

4 Design

Our prototype focuses on a specialized input generator for XML parsers. The core of our design involves crafting malformed XML packets using ML techniques, employing the AFL as our choice of fuzzing engine due to its advanced functionality and adaptability. AFL is widely regarded as one of the best multi-purpose fuzzers available. It combines extensive capabilities with a compact code base, allowing for easy modification and extension. AFL++, an advanced version of AFL, incorporates numerous scientific advances that make it one of the most powerful fuzzers currently available. These enhancements, however, also increase the complexity of the codebase, making custom extensions to AFL++ more difficult to implement. Therefore, we choose AFL as the basis for the experimental setup. This allows us to focus on the interface between ML and the fuzzer with little modification on the AFL side. AFL orchestrates the fuzzing operation by methodically feeding inputs to the ML parser, monitoring its behavior, and recording any crashes, hangs and anomalies.

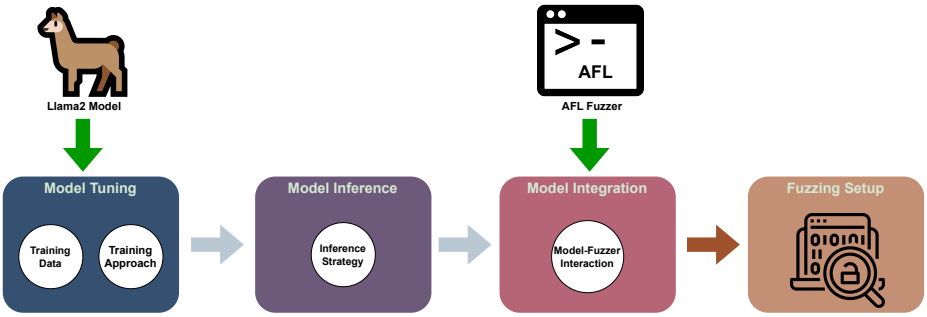

Fig. 1. Concept overview of our approach

On the machine learning side, we require an LLM that can effectively handle context-free grammars and generate complex input. With Llama2 13B, we choose a well-known and freely available LLM [17], as its ability to efficiently parse and generate complex structures perfectly matches our goal of systematically generating malformed XML packages.

We focus on XML, because its human-readable format allows immediate visual verification of results. This also fits well with our ML methods, allowing the systematic production of malformed packets to test the robustness of the parser. In addition, the ubiquity of XML provides a wealth of test cases, and its structured nature further aids the verification of results, making it easier to identify parser weaknesses. Since XML can be represented as a formal grammar, it furthermore allows us to compare our results with a grammar-based fuzzer, providing a benchmark for evaluating the effectiveness of our approach. Figure 1 illustrates the concept of our solution, necessitating two principal inputs: An LLM, specifically Llama2, and the AFL fuzzing tool. Our method consists of three phases:

- **Model Tuning:** Initially, we refine the ability of the model to process the input grammar of the target program by selecting an optimal dataset and training method. This phase results in an LLM tailored to process input structures relevant to the target program, enhancing its efficiency in generating meaningful input variations.
- **Model Inference:** Following the tuning, this phase involves deploying the model to produce test inputs for the fuzzing process. The focus here is on employing a strategic inference mechanism that enables the generation of diverse test cases.
- **Model Integration:** This final step ensures the model works smoothly with the fuzzing tool. We outline a straightforward method for the LLM to feed inputs directly into the fuzzing process, aiming for seamless collaboration between the two.

This structured approach results in a comprehensive fuzzing environment, encompassing an LLM proficient in generating targeted input samples and an AFL instance refined to leverage these samples effectively.

4.1 Dataset

We gather a dataset consisting of both malicious and benign XML files. For the malicious files, we obtain 56 XML files from the Exploit Database[4], primarily containing XML External Entity (XXE) injection payloads that exploit real-world application vulnerabilities. For the benign files, we choose the KIT Motion-Language Dataset [12] as the source and randomly select 100 XML files from there to keep benign and malicious samples roughly in balance.

While this seems to be a fairly small set of training data, it is a challenging task to obtain a larger variety of distinct malicious XML files. This, however, should not be a problem, as Llama2 is pre-trained on large amounts of publicly available XML files found on the Internet, providing a strong foundation for the model's understanding of the XML structure.

4.2 Training Approach

The training phase involves obtaining data aligned with the goals of the project, where the generative LLM is utilized to supply AFL with a balanced mix of benign and malicious samples, ensuring a thorough evaluation of the functionality of the target program. Following dataset preparation, we proceed to determine the optimal model tuning method.

Although Fine-Tuning requires significant computational resources, Prompt-Tuning offers a more cost-effective alternative, as only a portion of the model is modified. Given these considerations, we prioritize Prompt-Tuning for its efficacy, with Fine-Tuning serving to establish a robust comparative baseline.

4.3 Inference Strategy

Inference strategies for generating fuzzing samples broadly fall into two primary categories: sampling-based strategies, that explore the input space using randomness, and deterministic strategies, that use a predefined algorithm or heuristics. Deterministic strategies, which produce the same sequences for given tokens and probability tuples, lack the variability necessary for effective fuzzing. Such predictability, coupled with scalability issues related to memory and execution time, renders deterministic strategies impractical for our application.

Our chosen approach is the top-k sampling method, which belongs to the category of sampling-based strategies. The adaptability of top-k sampling makes it ideal for generating a wide range of fuzzing inputs, facilitating a more comprehensive evaluation of the vulnerabilities of the target program. This technique

[4] `https://www.exploit-db.com/`

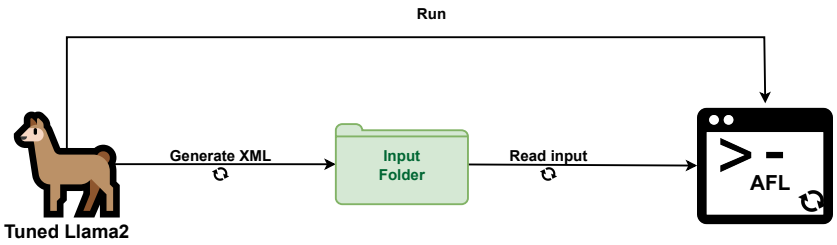

Fig. 2. LLM Integration into the Fuzzing Test

involves selecting the top-k most likely tokens at each step, introducing randomness into the process to ensure a varied set of generated XML samples. The key advantage of this method is its ability to maintain a balance between generating diverse inputs and keeping the selection process efficient and manageable. This balance is crucial for fuzzing applications where both the diversity of inputs and the practicality of generating them are important.

4.4 Model Integration

With the LLM prepared for inference, we outline an integration strategy enabling the application of the model within the fuzzing tests. Unlike conventional tools that depend on predefined grammars, our approach benefits from directly incorporating generated XML samples into the fuzzing workflow.

This integration necessitates modifying AFL to accommodate the seamless transfer of generated samples, detailed further in section 5. As shown in Figure 2, the Llama2 model is activated to produce a predetermined number of XML samples. These samples are directly sent to an input directory, making them readily available for AFL processing. The operation is cyclic, with new samples generated and assessed by AFL throughout the duration of the fuzzing test.

The cycle begins with the LLM initiating AFL to start a fuzzing session that is scheduled to run continuously for 24 hours. The initial time required for the LLM calculations is not subtracted from the AFL fuzzing time, as this time can be calculated in advance. In real-world scenarios, testing time is often limited and typically begins as soon as the program is ready. After the 24-hour period, sample generation stops and AFL exits. This termination is managed either by a direct stop signal from the LLM or by a predetermined timeout mechanism set at the start of the test.

4.5 Feedback Loop

An integral component of our system, as depicted in Figure 3, is the feedback loop that enhances the performance of the LLM over time. During the fuzzing test, the LLM adapts based on the feedback of the fuzzer. Specifically, the feedback loop (illustrated by red arrows in the figure) analyzes AFL outcomes to pinpoint

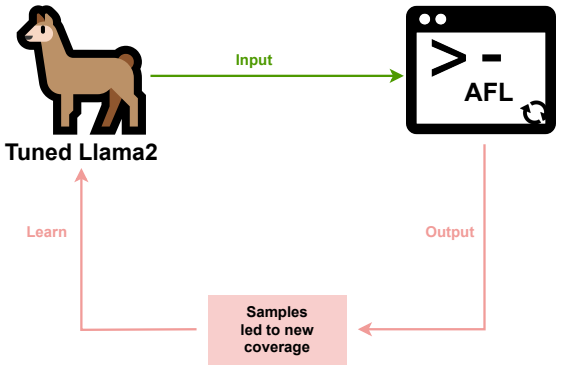

Fig. 3. Llama2 learns through a Feedback Loop

XML samples instrumental in uncovering new vulnerabilities or pathways in the target program. These samples are then reintegrated into the training routine of the LLM during regular Prompt-Tuning sessions. This allows the model to improve its process of generating examples for future tests. This iterative cycle allows the model to continuously refine its output, aligning more closely with the evolving landscape of potential vulnerabilities in the target software.

5 Implementation

We develop a Proof of Concept (PoC) leveraging the synergies between the 13B model of Llama2 and the AFL for an enhanced fuzzing framework. Our PoC, hosted on GitHub[5], demonstrates the practical application of integrating an LLM with a leading fuzzing tool to uncover software vulnerabilities through structured XML inputs.

5.1 Model Training and Integration with AFL

Fine-Tuning To begin our implementation, we perform a full Fine-Tuning of the Llama2 model, which was specifically chosen for its balance between computational efficiency and model performance.

This Fine-Tuning process is performed over three epochs with a learning rate of 0.003. This learning rate value is determined through empirical testing to strike a balance between rapid model adaptation and the risk of overfitting. Fine-Tuning is performed using a curated dataset as described in section 4.1.

Prompt-Tuning We also use a Prompt-Tuning approach using the Transformers library, training the model on the same dataset mentioned in section 4.1.

[5] `https://github.com/IbraMhiri/LLaMa-Fuzzer`

To compare the results, we use the 3072 and 4096 tokenization settings. With 3072 tokenization, the model processes queries up to 3072 tokens long, while with 4096 tokenization it processes queries up to 4096 tokens long. This difference affects the contextual information available for output generation.

5.2 Continuous Data Integration Mechanism

Our enhancements to the AFL framework includes the implementation of a continuous data integration mechanism, distinguishing our approach from traditional fuzzing methods. This mechanism allows AFL to dynamically apply new test cases throughout the fuzzing process, rather than relying on a static set of inputs. By modifying the `read_testcases()` function of AFL to periodically scan a predefined input folder, our system ensures a constant influx of fresh, model-generated XML samples for testing. To prevent redundancy, we incorporated a function to track and skip previously scanned samples, optimizing the efficiency of the fuzzing process. We have also configured AFL to store the test cases responsible for detecting new unique paths within the target binary, as well as any hangs or crashes. These test cases are not only stored in the internal AFL seed pool, but also in a predefined folder. This folder serves as a repository for cases that will later be used by the feedback loop mechanism to further improve our model.

5.3 Optimization Technologies

To further improve the scalability and efficiency of our implementation, we integrate optimization technologies such as Accelerate [5], DeepSpeed [15] and Zero [14, 15]. These tools are instrumental in enabling the rapid training and execution of the LLM on hardware configurations that are accessible to a broad range of researchers. In addition, our implementation takes advantage of technologies such as Accelerate, which provides optimization techniques to improve the efficiency and scalability of training and using large neural network models.

5.4 Dynamic Feedback Loop

Our approach is to use a dynamic feedback loop mechanism to review the results of each fuzzing iteration.

First, AFL reads the XML files generated by the LLM and starts fuzzing. Then the analysis phase begins: we evaluate the effectiveness of different XML patterns to identify those that successfully reveal new execution paths or cause program crashes. Patterns (XML files) that contribute to new code coverage are then fed back to the LLM. Conversely, patterns that do not add code coverage are retained in the AFL seed pool, but are not fed back to the LLM. The LLM then runs Prompt-Tuning on these samples to improve its performance.

The LLM is then Prompt-Tuned using these samples to improve its performance. Following the Prompt-Tuning process, the LLM generates new XML

files that AFL uses as input for further fuzzing. This feedback loop allows us to prioritize XML patterns that have proven effective in exploring new paths or triggering crashes. This iterative process facilitates continuous improvement in the effectiveness of the fuzzing approach over time.

In the meantime, AFL works as usual. However, as new XML files are received from the LLM, they are read and added to the seed pool for further fuzzing. AFL then resumes normal operation, actively looking for new code coverage.

6 Evaluation

We used the developed PoC as described in section 5 to evaluate our approach.

The development environment and hardware played a crucial role in determining the results of this research, given the intensive computational requirements of training and using LLMs. We used a server equipped with a NVIDIA A100 GPU with 80 GB of VRAM. The server also had 720 GB of RAM, 2.03 TB of storage and was powered by an AMD EPYC 75F3 CPU.

To ensure comparability between runs, we ran each fuzzing session for 24 hours, allowing full access to GPU resources while restricting the fuzzing process to a single CPU core (default setting of AFL). In real-world scenarios, resources are typically not the bottleneck for fuzzing. Running AFL in parallel on multiple cores does not scale linearly because of the need to share results and data, which introduces overhead.

For our evaluation, we tested the fuzzing setup we developed against several target programs, including libxml2 and TinyXML-2.

6.1 Evaluation Metrics

The following metrics provided by AFL have been used for the evaluation:

- Total Paths Found: This metric represents the total number of unique paths found within the target program during the fuzzing test.
- Crashes: These are unique test cases that cause fatal errors in the tested program, such as `SIGSEGV`, `SIGILL`, `SIGABRT`, and others.
- Hangs: These are unique test cases that cause the unit under test to time out. In AFL, the default time limit before a test-case is classified as a hang is 1 second.

For our evaluation we rely on two widely used XML parsers. A common method of evaluating the effectiveness of a fuzzer is to introduce deliberate bugs into the tested program and measure how many of them are detected by the fuzzer. However, we take an alternative approach by testing real-world programs, as intentionally introduced bugs are more obvious and lack the subtlety often found in accidentally introduced bugs. Consequently, if no bugs are found – often because the program has been thoroughly tested and simpler bugs have already been patched – code coverage (total paths found) becomes the most reliable metric. It serves as an indicator of how deeply the fuzzer has explored the program.

6.2 Experimental Approaches

To evaluate our prototype, we conducted experiments using different approaches, each of which yielded different results:

- AFL Fuzzing: This approach uses the native functionality of the AFL fuzzing tool, with the training data used for the LLM as input.
- Pre-Trained LLM Fuzzing: This method uses the original pre-trained Llama2 model (13B version) as input provider to AFL.
- Fine-Tuned LLM Fuzzing: This method uses a Fine-Tuned version of Llama2 to provide input to AFL.
- Prompt-Tuned LLM Fuzzing: This approach uses a Prompt-Tuned Llama2 model.
- Feedback Loop LLM Fuzzing: The final approach, our PoC, uses a Prompt-Tuned Llama2 model to provide input samples to AFL. In addition, this model undergoes real-time Prompt-Tuning during the fuzzing test using feedback from the fuzzer, specifically samples that lead to the discovery of new paths.
- Nautilus[6] and AFL Fuzzer combined: This approach uses Nautilus 2.0, a coverage-guided, grammar-based fuzzer, in combination with AFL as a baseline. AFL synchronizes with Nautilus, allowing AFL to import Nautilus input, but not vice versa. We used the example of a XML structure that comes with nautilus (`grammar_py_exmaple.py`).

These approaches allow us to compare the effectiveness and efficiency of different methods in the context of our fuzzing framework.

6.3 Experimentation Results

The plots in Figure 4 illustrate the number of paths detected by the fuzzer within libxml2 and TinyXML-2, respectively, over 24 hours. The LLM enhanced fuzzing methods outperform the traditional AFL approach, as shown in the figure. All AFL variants integrated with an LLM show superior performance compared to using AFL alone, with tests using Prompt-Tuned LLMs showing significant improvement. In the following, we discuss and interpret our results.

For libxml2, the traditional AFL detected 1698 paths, while LLM-enhanced AFL detected 10705 paths. The Prompt-Tuned LLM 4096 model detected 8203 paths, and the Prompt-Tuned LLM 3072 model detected 11290 paths. The Fine-Tuned LLM model detected 6719 paths, while the grammar-based fuzzer Nautilus combined with AFL detected 4826 paths.

For TinyXML-2, AFL detected 411 paths, while LLM-enhanced AFL detected 894 paths. The Prompt-Tuned LLM 4096 model detected 4745 paths, and the Prompt-Tuned LLM 3072 model detected 5000 paths. The Fine-Tuned LLM model detected 924 paths, while the grammar-based fuzzer Nautilus combined with AFL detected 1396 paths.

[6] `https://github.com/nautilus-fuzz/nautilus`

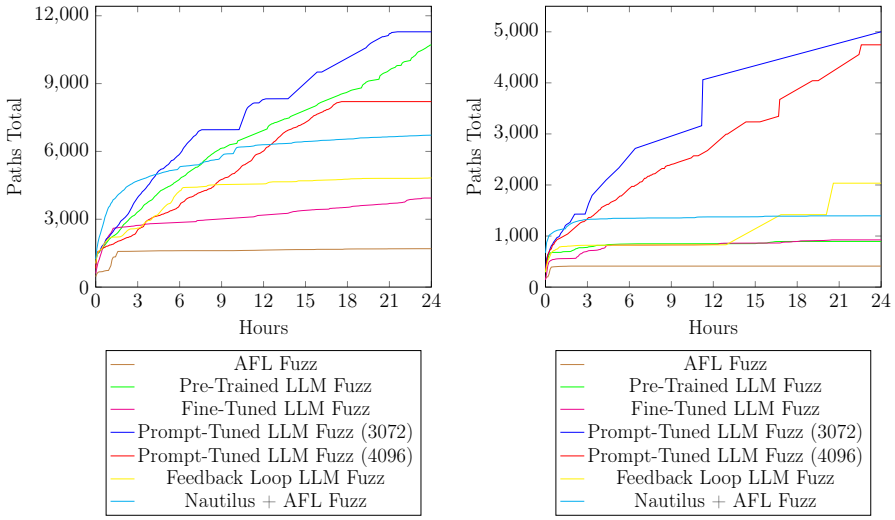

Fig. 4. Total path coverage of the different approaches in libxml2 (left) and TinyXML-2 (right) over 24 hours

The Fine-Tuned model, however, shows relatively poor results compared to other LLM-based tests. This prototype has undergone a minimal amount of training (only three epochs), which is not enough to learn the required grammar, but was directly determined by the limitations of our hardware resources.

Conversely, fuzzing approaches that use Prompt-Tuned Llama2 models, such as Prompt-Tuned and feedback loop models, outperform the pre-trained model. The increase in history provides Llama2 with more information to compute the most likely next tokens. Prompt-Tuned models are adept at learning new XML patterns, especially from malicious examples, thereby improving results and coverage. Adjusting the context length of Prompt-Tuned LLMs can further optimize performance, however considering a bigger context length can result in a worse performance as shown in Figure 4. Even though we could not derive a definite reason behind the deviation between the performance of the Prompt-Tuned LLM 4096 model and the Prompt-Tuned LLM 3072 model, our assumption is that using a bigger context length, can result in an overhead, which can introduce a considerable impact on the performance of the model over the 24 hours of the test period.

In our fuzzing tests, the combination of AFL with a Prompt-Tuned LLM using a context length of 3072 showed superior performance to all other methods when evaluated against both libxml2 and TinyXML-2.

Comparatively, the feedback loop model shows slightly fewer detected paths than the Prompt-Tuned model. This is due to the learning overhead of the feedback loop model, which periodically undergoes a tuning process using XML samples detected by the fuzzer.

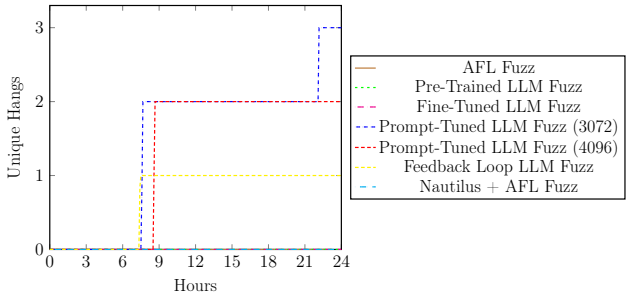

Fig. 5. Found hangs in libxml2 for the different approaches over 24 hours.

Figure 5 shows the number of hangs that occurred during the 24 hours of fuzzing libxml2. Again, the Prompt-Tuned fuzzing approaches outperformed the others, finding up to 3 unique hangs during the test period. However, it is worth mentioning that our approach, feedback loop fuzzing, follows right behind with one hang found. Standard AFL fuzzing, pre-trained LLM fuzzing, and Fine-Tuned LLM fuzzing did not find any hangs during the test. It is also worth noting that during a 24-hour fuzzing session targeting TinyXML-2, no hangs were detected regardless of the approach used. Also, no crashes were detected in either of the two programs tested.

Model	Pre-Trained	Fine-Tuned	Prompt-Tuned	Feedback Loop
Total number of Samples per Run	6060 Samples	980 Samples	13980 Samples	2390 Samples
Sample generation time	14.04sec	89.49sec	5.99sec	-

Table 1. Number of Generated Samples per Fuzzing Test

With the goal to better understand the potential direct factors that lead to the results presented in the previous paragraphs, we considered observing details related to samples generating. Table 1 represents the total number of generated samples per run as well as the required time to generate a single sample using the different LLM-based approaches that we introduced in our work. As we can notice here, the generation time of samples can significantly vary depending on the model type, which leads to a remarkable deviation within the total number of resulting samples. The original pre-trained Llama2 model generated 6060 samples within 24 hours, during which our test was running. On the other hand, the Fine-Tuned version generated only 980 samples. This is due to the complexity and extra layer that traditional Fine-Tuning adds to the model, which has a significant impact on the time required to generate a sample. The Prompt-Tuned (3072) model was the best performing model with 13980 samples. This result demonstrates the effectiveness of the Prompt-Tuning approach in improving the suitability and performance of the model for a specific

use case without adding complexity or extra layers. However, the feedback loop showed suboptimal performance in terms of samples generated. This is primarily due to our design, where the model periodically pauses sample generation to undergo a training phase.

7 Limitations and Future Work

The approach of combining a fuzzer with an LLM proves to be significantly more effective than using the fuzzer alone. In our study, we show that this hybrid approach can also yield superior results compared to traditional grammar-based fuzzers.

Traditional fuzzers rely heavily on mutation to explore new program paths, which often requires considerable luck to stumble upon meaningful variations. Even when fuzzers randomly generate new input, their inspiration often comes from the existing data initially provided to the fuzzer. In contrast, LLMs can generate arbitrarily complex XML files based on learned patterns from existing data, allowing them to penetrate deeper into the program logic directly.

These tests initially focused on XML, which is well suited for representation using grammar-based methods. In practice, however, we can test programs of arbitrary complexity.

The Prompt-Tuned LLM fuzzing approach emerged as the most successful method in our evaluation. Its effectiveness underscores the potential of integrating language models into fuzzing techniques to improve testing results and vulnerability detection.

When considering the strengths and limitations of using XML as our target domain, several factors come into play. The structured format and well-defined syntax of XML make it an ideal candidate for fuzzing, allowing for the systematic generation of test cases to assess parser robustness. However, the complexity of XML can also present challenges, especially when dealing with nested or deeply nested elements. In addition, XML parsers can vary in their handling of edge cases and error conditions, requiring extensive testing to identify potential vulnerabilities.

One of the key advantages of the Prompt-Tuned LLM fuzzing approach is its independence from explicit knowledge of the input grammar. Unlike traditional fuzzing methods that require a predefined grammar for input generation, Prompt-Tuned LLM fuzzing leverages the ability of the language model to implicitly learn and understand the grammar. This eliminates the need for manual grammar specification, streamlining the fuzzing process and enabling effective testing even in cases where the grammar is not accessible.

Furthermore, the adaptability of the approach extends beyond XML parsing to other applications, such as PDF parsing, that isn't context free. By leveraging the ability of the language model to reproduce and adapt complex structures, Prompt-Tuned LLM fuzzing provides a versatile solution for automated testing across different software systems.

In conclusion, the success of the Prompt-Tuned LLM fuzzing approach highlights the potential of integrating language models into fuzzing methodologies. By harnessing the power of natural language understanding, this approach offers a novel and effective means of automated testing that can be applied to diverse domains beyond XML parsing.

While this research has yielded positive results, it is important to acknowledge several limitations that have affected the results and interpretations. These limitations and challenges are listed below:

- Meanwhile, Llama3.1 has been released, which may offer improved results, but this work focuses on demonstrating feasibility rather than achieving the best performance with the fastest LLM. In addition, there are now more code-specific models such as Code Llama. However, we believe that standard LLMs are more appropriate for our data generation purposes.
- The choice of language models was constrained by the hardware described in section 5. Experimenting with larger language models for fuzzing tests holds the potential for better results, as they have more tunable parameters and a better understanding of context and languages.
- The proof of concept used 56 malicious and 100 benign XML examples. Exploring different data types and ratios of malicious to benign instances could improve the effectiveness of the proposed approach.
- The top-k sampling-based strategy, chosen to minimize sample generation time, warrants further investigation and optimization. Balancing generation time and sample quality through parameter tuning, taking into account other inference strategies, could further improve the results.
- The feedback learning loop, as discussed in Section 6, affects the number of samples generated during fuzzing tests. Optimization through asynchronous processes for learning and generation phases could improve this approach.

Looking forward, this research also opens up opportunities for further investigation and possible research directions. Some suggested topics arising from this work include:

- Exploring AI Alternatives: Investigating the viability of alternative AI models, such as Generative Adversarial Networks (GANs) and Sequence to Sequence (Seq2Seq) models, to improve the efficiency of fuzzing tests. In addition, the application of Low-Rank Adaptation (LoRA) could improve the results of Fine-Tuning LLMs with small datasets.
- LLM-Enhanced Mutation-Based Fuzzing: Examining the potential of LLM-based methods to augment and enhance the mutation operations performed by advanced fuzzing tools such as AFL. This examination could include developing techniques to use the contextual understanding provided by LLMs to more effectively guide mutation strategies, leading to improved test case generation and vulnerability discovery.
- Domain-Specific Adoption: Researching the applicability of the proposed solution in domains beyond XML-based applications, particularly in programs

that deal with input governed by specific grammars that may challenge traditional fuzzing methods. It is important to consider how to extend our approach and in what format these adaptations can be made effectively, emphasizing the need to explore appropriate formats and methodologies for integration into different domains. This adoption requires the adaptation of the tuning phase of our approach by using appropriate samples of the target domain, for instance learning PDF files.

As outlined in Section 8, this research addresses various aspects of LLM-based fuzzing and represents a promising direction for future research. The evaluation presented in Section 6 demonstrates the potential of even a simple pre-trained Llama2 model to contribute to automated testing, thereby increasing the efficiency of fuzzing tests for security-critical applications such as XML parsers. Future work could also address the limitations outlined above: Opting for the latest version of Llama, optimizing the top-k sampling strategy, using more hardware resources and training on a larger dataset could further evaluate the applicability of our approach.

8 Conclusion

In this work we explored the potential of improving fuzzing tests using LLMs, with a focus on applications that parse XML. Our method, introduced in section 4, revolves around Fine-Tuning and Prompt-Tuning a pre-trained LLM to capture the nuances of XML grammar and tailor it to the task of generating suitable XML samples for fuzzing tests. This involved training the model on a combination of normal and malicious data through Prompt-Tuning.

To demonstrate these concepts, we developed a PoC and made it available on GitHub to facilitate public access and collaborative improvement. Building upon the pre-trained Llama2 model, we applied our method and integrated the enhanced model into AFL, a state-of-the-art fuzzing tool.

Our experimental phase primarily examined the XML parsers libxml2 and TinyXML-2, revealing notable enhancements in the efficiency of AFL when incorporating LLMs. The evaluation demonstrated significant advancements in the discovery of new paths and increased bitmap coverage over set periods. The study not only highlights these key outcomes but also delves into several technical specifics encountered throughout the research.

Our fuzzing tests showed that the use of AFL in conjunction with a Prompt-Tuned LLM with a context length of 3072, outshone other tested methods in terms of code path coverage for both libxml2 and TinyXML-2. Noteworthy is the ability of our method to uncover three distinct unique hangs in libxml2 during fuzzing, underscoring the potential of integrating LLMs into fuzzing workflows for improved testing efficacy.

Acknowledgments. This publication is based on the research project SofDCar (19S21002), which is funded by the German Federal Ministry for Economic Affairs and Climate Action.

A Appendix

A central aspect of the inference evaluation is runtime. The required time for each model to generate an XML sample was recorded and represented in this part of the evaluation.

$Top-k$ Value	5	25	50	150	250
Pre-Trained LLM	13,12sec	14,04sec	14,34sec	14,56sec	14,68sec
Fine-Tuned LLM	89,82sec	89,49sec	90,67sec	89,52sec	91,35sec
Prompt-Tuned LLM	5,05sec	5,99sec	6,5sec	6,97sec	7,32sec

Table 2. Top-k variation impact on XML Generation Time

The table 2 shows the required time by the model to generate an XML sample in seconds for several $top-k$ values. A high value can impact the generation time of XML samples (with slight variations). However, aside from this aspect, an interesting behavior shown in this table is that the generation time of XML samples can significantly vary depending on the model type (fine-tuned, prompt-tuned, or pre-trained). This table shows that the prompt-tuned LLM is the fastest model, whereas the fine-tuned one is very slow.

References

1. Ackerman, J., Cybenko, G.: Large language models for fuzzing parsers (registered report). In: Proceedings of the 2nd International Fuzzing Workshop. p. 31–38. FUZZING 2023, Association for Computing Machinery, New York, NY, USA (2023), https://doi.org/10.1145/3605157.3605173
2. Deng, Y., Xia, C., Yang, C., Zhang, S., Yang, S., Zhang, L.: Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 830–842. ICSE '24, IEEE Computer Society, Los Alamitos, CA, USA (apr 2024), https://doi.org/10.1145/3597503.3623343
3. Deng, Y., Xia, C.S., Peng, H., Yang, C., Zhang, L.: Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models (2023), https://doi.org/10.48550/arXiv.2212.14834
4. Godefroid, P., Peleg, H., Singh, R.: Learn&fuzz: Machine learning for input fuzzing. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 50–59. IEEE Computer Society, Los Alamitos, CA, USA (2017), https://doi.org/10.1109/ASE.2017.8115618
5. Gugger, S., Debut, L., Wolf, T., Schmid, P., Mueller, Z., Mangrulkar, S., Sun, M., Bossan, B.: Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate (2022)
6. Hu, J., Zhang, Q., Yin, H.: Augmenting greybox fuzzing with generative ai (2023), https://doi.org/10.48550/arXiv.2306.06782

7. Lemieux, C., Inala, J.P., Lahiri, S.K., Sen, S.: Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: Proceedings of the 45th International Conference on Software Engineering. p. 919–931. ICSE '23, IEEE Press, Los Alamitos, CA, USA (2023), https://doi.org/10.1109/ICSE48619.2023.00085
8. Lester, B., Al-Rfou, R., Constant, N.: The power of scale for parameter-efficient prompt tuning (2021), https://doi.org/10.48550/arXiv.2104.08691
9. Liu, D., Metzman, J., Chang, O.: Ai-powered fuzzing: Breaking the bug hunting barrier. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html (8 2023), google Open Source Security Team
10. Meng, R., Mirchev, M., Böhme, M., Roychoudhury, A.: Large language model guided protocol fuzzing. In: Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS). The Internet Society, Reston, VA, USA (01 2024), https://doi.org/10.14722/ndss.2024.24556
11. Pathak, H.: Parameter-efficient fine-tuning (peft) and how it's different from fine-tuning. https://medium.com/@harshnpathak/parameter-efficient-fine-tuning-peft-and-how-its-different-from-fine-tuning-3f6b95c73bac (7 2024)
12. Plappert, M., Mandery, C., Asfour, T.: The KIT motion-language dataset. Big Data **4**(4), 236–252 (dec 2016), http://dx.doi.org/10.1089/big.2016.0028
13. von Platen, P.: How to generate text: using different decoding methods for language generation with transformers (03 2020), https://huggingface.co/blog/how-to-generate
14. Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., He, Y.: Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21, Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3458817.3476205
15. Rasley, J., Rajbhandari, S., Ruwase, O., He, Y.: Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. p. 3505–3506. KDD '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3394486.3406703
16. Salem, H.A.A., Song, J.: A review on grammar-based fuzzing techniques. International Journal of Computer Science and Security (IJCSS) **13**(3), 114 – 123 (June 2019), http://www.cscjournals.org/library/manuscriptinfo.php?mc=IJCSS-1481
17. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C.C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M.A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T.: Llama 2: Open foundation and fine-tuned chat models (2023), https://arxiv.org/abs/2307.09288

18. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al.: Llama 2: Open foundation and fine-tuned chat models (2023), https://doi.org/10.48550/arXiv.2307.09288
19. Wang, Y., Jia, P., Liu, L., Huang, C., Liu, Z.: A systematic review of fuzzing based on machine learning techniques. PLOS ONE **15**(8), 1–37 (08 2020), https://doi.org/10.1371/journal.pone.0237749
20. Wu, Z.: A Study of Grammar-Based Fuzzing Approaches. Master's thesis, California Polytechnic State University (2022), https://doi.org/10.15368/theses.2022.69
21. Xia, C.S., Paltenghi, M., Tian, J.L., Pradel, M., Zhang, L.: Fuzz4all: Universal fuzzing with large language models (2024), https://doi.org/10.48550/arXiv.2109.05687
22. Xu, R., Luo, F., Zhang, Z., Tan, C., Chang, B., Huang, S., Huang, F.: Raise a child in large language model: Towards effective and generalizable fine-tuning (2021), https://doi.org/10.48550/arXiv.2109.05687
23. Yang, C., Deng, Y., Lu, R., Yao, J., Liu, J., Jabbarvand, R., Zhang, L.: White-box compiler fuzzing empowered by large language models (2023), https://doi.org/10.48550/arXiv.2310.15991
24. Yang, C., Zhao, Z., Zhang, L.: Kernelgpt: Enhanced kernel fuzzing via large language models (2023), https://doi.org/10.48550/arXiv.2401.00563
25. Yemme, A., Garani, S.S.: A scalable gpt-2 inference hardware architecture on fpga. In: 2023 International Joint Conference on Neural Networks (IJCNN). pp. 1–8. IEEE, IEEE Computer Society, Los Alamitos, CA, USA (2023), https://doi.org/10.1109/IJCNN54540.2023.10191067
26. Yong, Z.X., Schoelkopf, H., Muennighoff, N., Aji, A.F., Adelani, D.I., Almubarak, K., Bari, M.S., Sutawika, L., Kasai, J., Baruwa, A., Winata, G., Biderman, S., Raff, E., Radev, D., Nikoulina, V.: Bloom+1: Adding language support to bloom for zero-shot prompting (Jul 2023), https://aclanthology.org/2023.acl-long.653
27. Zhang, H., Rong, Y., He, Y., Chen, H.: Llamafuzz: Large language model enhanced greybox fuzzing (2024), https://arxiv.org/abs/2406.07714