ECE-GY 7123 / CS-GY 6953 / Deep Learning - Fall '25
Final Project Report
Project: Fine-Tuning SLMs for Targeted XML Fuzzing

Thanh Do
New York University
qd2121@nyu.edu

Abstract

Fuzzing structured data formats like XML/XSD is a persistent challenge: traditional bit-flipping mutation destroys syntax/hierarchical validity, while grammar-based approaches require labor-intensive formal definitions. This project investigates the use of Small Language Models (SLMs), specifically `Qwen3-0.6B`, as *learned fuzzers*. Unlike Large Language Models (LLMs) which are aligned to correct user errors, we hypothesize that SLMs trained on the W3C XML Schema Test Suite without instruction prompting will learn an "approximate grammar"; just sufficient to pass the target parser's syntax checks, but prone to "hallucinations" (e.g., inventing plausible but invalid attributes) that act as semantic mutations. We employ Low-Rank Adaptation (LoRA) to fine-tune the model and evaluate its effectiveness by measuring code coverage on the `python-xmlschema` library. While our training pipeline leaves a lot to be desired, some preliminary results during our experiments show that this "learned hallucination" approach shows promising behaviors that's worth investigating further.

1 Introduction

Software libraries that parse structured data (XML, JSON, PDF) are critical infrastructure, yet they are notoriously difficult to automatically test ("fuzz"). Traditional fuzzers often fail to penetrate deep validation logic because bit-level mutations destroy the strict syntax required for parsing. Conversely, grammar-based approaches require labor-intensive formal definitions that may restrict the generation of useful edge cases.

From these problems, we propose using a Small Language Model (SLM) as a generative fuzzer as an alternative option. By leveraging transfer learning, we aim to bypass the need for manually written context-free grammars. By relying on the *hallucinating* and *contextual forgetfulness* behaviors of current language models, we argue that these attributes make *good* fuzzers for structural file formats that are close to natural language such as XML.

Our contributions include:

- We adapt a pre-trained Small Language Model (`Qwen3-0.6B`) for structural fuzzing using Low-Rank Adaptation (LoRA), enabling efficient local generation.
- We evaluate a "No-Prompt" training strategy on the W3C XML Schema Test Suite that treats XML as raw code, encouraging the model to learn an approximate grammar while retaining stochastic hallucination behaviors.
- We test our approach on the `python-xmlschema` library, demonstrating higher code coverage compared to zero-shot baselines.

2 Background and Related Work

2.1 Fuzz testing (fuzzing)

Fuzz testing, or fuzzing, is an automated software testing technique that involves injecting invalid, unexpected, or stochastically generated inputs into a program to uncover implementation bugs, crashes, and security vulnerabilities.

Traditional *mutational fuzzers* like Radamsa[1] or LibAFL[2] modify input files without structural awareness, which breaks the rigid syntax of formats like XML immediately. This results in "shallow" testing: the parser rejects the input right after the beginning, and the deep validation logic remains untested, even with the evolutionary test case selection approach used in AFL.

[1] https://gitlab.com/akihe/radamsa
[2] https://github.com/AFLplusplus/LibAFL

Generational fuzzers like `libprotobuf-mutator`[3] or Peach[4] are needed to test such targets. However, these approaches rely on manually written context-free grammars, which suffers from two main problems: they are *labor-intensive to create* and often *too perfect to introduce bug-prone behaviors*.

2.2 Language Models for Automated Testing

Deep learning models, particularly Large Language Models (LLMs) trained on code (e.g., Qwen), offer another approach that bridges the gap between chaotic mutation and rigid grammars.

Unlike formal grammars which enforce strict validity, these models learn an *approximate grammar*: a continuous probability distribution over tokens based on vast training corpora. This capability allows them to generate inputs that respect the high-level syntax required to pass a parser (e.g., matching XML tags) without explicit human programming. Furthermore, because they are probabilistic, they naturally introduce *hallucinations*, which should act as semantically rich mutations that are difficult to encode in manual grammars. This should help ease the tension between conflicting learning and fuzzing goals, as formulated in (Godefroid et al., 2017).

Recent works (Deng et al., 2023) (Xia et al., 2024) have demonstrated that such models can effectively automate test case generation, learning to produce edge cases that explore deep program states inaccessible to purely random fuzzers.

However, the deployment of massive models reveals a critical bottleneck in the context of fuzzing campaigns.

2.3 High-Throughput Fuzzing with Small Language Models

Fuzzing behaves similarly like search problems (Donaldson et al., 2025): volume is key. Discovering deeply buried vulnerabilities often requires generating and executing millions of inputs to trigger rare race conditions or edge cases. While LLMs excel at reasoning, they suffer from high inference latency and prohibitive computational costs that stifle the necessary throughput. A 70-billion parameter model running on a remote API cannot match the iteration speed of a local fuzzer generating thousands of inputs per second.

We argue that the broad "world knowledge" possessed by LLMs, such as history, math, or natural language nuances, is largely redundant for the specific task of structural fuzzing. Testing a library like `python-xmlschema` requires a deep understanding of syntax constraints, not general reasoning. This creates a compelling opportunity for Small Language Models (SLMs). By restricting the domain to specific formal grammars, we hypothesize that SLMs (sub-4B models) can be fine-tuned to master the target syntax while running locally with the high throughput required to effectively explore the state space.

2.4 Low-Rank Adaptation (LoRA)

Low-rank adaptation (Hu et al., 2021) freezes the pre-trained model weights and injects trainable rank decomposition matrices into the layers of the Transformer architecture. This allows us to fine-tune a 4B parameter model with minimal GPU memory, making it feasible for our course constraints.

3 Problem Statement and Goals

3.1 Problem Description

The primary challenge in fuzzing structured formats like XML is that random mutations destroy syntactic validity (e.g., malformed tags), causing inputs to be rejected by the parser before they can test deeper logic. Conversely, grammar-based tools are often too rigid, requiring manual effort to define and failing to produce "creative" invalid inputs.

3.2 Objectives and Scope

In this project, we will perform test case generation and evaluation on the `python-xmlschema` library, a comprehensive validator for the W3C XSD standard.

Our goal is to maximize **statement (basic-block) code coverage** in the `python-xmlschema` library, by generating inputs that are "valid enough" to pass the parser to a certain stage, but "interesting enough" to trigger edge cases in the validator.

We select `Qwen3-0.6B` (Qwen Team, 2025), a model pre-trained heavily on code, and fine-tune it via LoRA (Hu et al., 2021) on the W3C XML Schema Test Suite (Thompson et al., 2004).

We hypothesize that a LoRA-tuned model will achieve lower perplexity on valid W3C data than

[3] https://github.com/google/libprotobuf-mutator

[4] https://peachtech.gitlab.io/peach-fuzzer-community/

the base model. Consequently, we expect the fine-tuned model to generate inputs that survive the parsing stage at a higher rate, thereby exposing the validation engine to "hallucinated" semantic structures.

We rely on three core arguments for this approach:

1. **The Approximate Grammar Hypothesis:** Neural networks learn continuous probability distributions over tokens. This allows them to approximate the rigid rules of XML (e.g., matching tags) without explicit programming.

2. **Efficiency of SLMs:** As described in the Background section, fuzzing is a volume game. Generating thousands of test cases requires high throughput. SLMs can run locally in a tight loop, whereas massive LLMs (like GPT-4) are API-bound and too slow and costly for fuzzing campaigns.

3. **Hallucination as a Feature:** In chat applications, hallucination is a defect. In fuzzing, it is a mutation operator. A model that "hallucinates" a non-existent XML attribute (e.g., `<xs:element recursive="true">`) generates a high-quality test case that tests how the library handles unexpected schema definitions.

4 Approach

4.1 Architecture

An overview of the whole fuzzing system can be found in Figure 1. The language model follows the Causal Language Modeling approach.

- **Base Model:** `Qwen/Qwen3-0.6B` (Qwen Team, 2025).
- **Adaptation:** Low-Rank Adaptation (LoRA) (Hu et al., 2021) on all layers.
- **Training Objective:** Minimize Cross-Entropy Loss on the W3C dataset.

4.2 Methods

A critical design decision in this work, which differs from our previous midterm report (Do, 2025) and the project proposal, is the *absence of natural language instructions*. We do not use prompts such as "Generate a valid XML file." Instead, we prompt the model with raw code prefixes (e.g., `<xs:schema`).

We justify this strategy based on the alignment variance between Chat and Code models. Instruction tuning (RLHF) typically optimizes models to be "helpful" and "correct" (Ouyang et al., 2022). If instructed to generate XML, an aligned model attempts to produce perfectly valid, standard code: effectively "auto-correcting" the very edge cases we wish to generate. By stripping the instructions and relying on raw Causal Language Modeling, we bypass safety filters and helpfulness biases, accessing the model's raw probability distribution. This maximizes the entropy of the output, encouraging the generation of the "long tail" of syntax: the weird, valid-but-rare structures necessary for effective fuzzing (Godefroid et al., 2017).

4.3 Implementation Details

4.3.1 Environment Details

The training and inference process for our model was conducted on Google Colab using a High-RAM A100 GPU with 80GB of VRAM and 168GB of RAM. The version of the Colab notebook runtime environment (and the corresponding libraries) was the latest as of this writing for this machine class (`2025.10`).

The `python-xmlschema` version under test is 4.2.0, which is the latest available at the time of writing.

4.3.2 Training Strategy

We used the `Trainer` class from the Hugging Face `Transformers` library[5]. Training parameters are as follows:

- **Batch Size:** 4
- **Gradient Accumulation Steps:** 4
- **Optimizer:** `adamw_torch`
- **Max New Tokens:** 256
- **Max Sequence Length:** 1024
- **Learning Rate:** 2×10^{-4}
- **LR Scheduler:** `linear` (by default)
- **Number of Epochs:** 15
- **Warmup steps:** 5

[5] https://huggingface.co/docs/transformers/en/main_classes/trainer

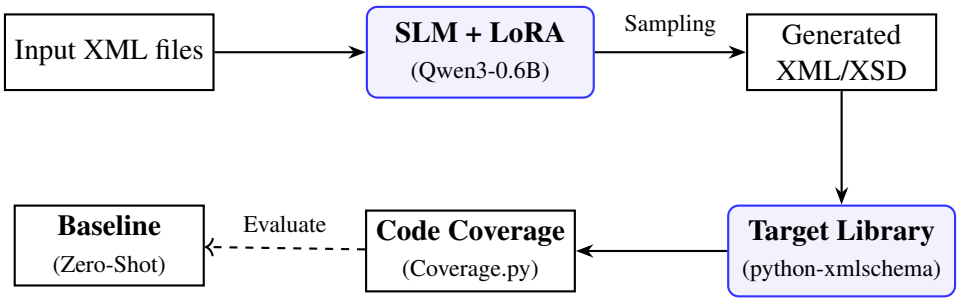

Figure 1: Proposed Fuzzing Pipeline. The LoRA-tuned model generates inputs which are fed into the target Python library. Execution traces are captured to calculate coverage metrics.

- **Weight decay:** 0.01

LoRA parameters are as follows:

- **LoRA Rank** (r)**:** 8
- **`lora_alpha`:** 16
- **`lora_dropout`:** 0.05
- **Target Layers:** All layers

Model generation parameters:

- **Temperature** T**:** 0.8
- **`top_p`:** 0.8
- **`top_k`:** 20

5 Data and Experimental Setup

5.1 Dataset

We curated a subset of the **W3C XML Schema Test Suite** (Thompson et al., 2004). Only `*.xsd` files are considered into the dataset. While we can also train with other types of XML files, for computational constraints, we decided to focus only on XSD.

- **Preprocessing:** Minification by removing comments and whitespaces between tags. No instruction prompting was used.
- **Sampling:** We used exactly 4,096 fully terminated XML samples from the dataset. We avoided using truncated samples in order to teach the model to close the tags properly.
- **Split:** 90% Training, 10% Validation (used for Perplexity monitoring).

5.2 Evaluation

We compare:

1. **Baseline:** `Qwen3-0.6B` (Zero-Shot).
2. **Experiment:** `Qwen3-0.6B` + LoRA (Fine-Tuned).

Metric: Our primary evaluation metric is the percentage of `python-xmlschema` code statements executed, measured using the `coverage.py`[6] tool. We prioritize statement coverage over simple input validity because effective fuzzing requires triggering deep error-handling logic, often accessible only through semantically invalid edge cases. Consequently, while we report the validity rate, it serves as a secondary, informative metric rather than the primary optimization objective.

5.3 Evaluation Protocol

We perform code coverage measurement on `python-xmlschema` by sampling 100 files from each model then instantiate the `XMLSchema`[7] class on each sample.

We found that the model tends to "ramble" after finished generating the XSD data (Figure 2), so we extract the first valid XSD root tag from the responses as a post-processing step.

To measure the "verbatim memorization" problem (discussed in Section 7 below), we calculate the Perplexity metric for the final model.

6 Results

6.1 Quantitative Results

The experimental results are shown in Table 1. We observe a concurrent increase in both `validity` and `coverage` for the fine-tuned model, providing

[6] `https://coverage.readthedocs.io/`
[7] `https://xmlschema.readthedocs.io/en/latest/api.html#xmlschema.XMLSchema`

```
<?xml version="1.0"?><xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
><xsd:element          name="doc"><xsd:complexType><xsd:choice><xsd:element
name="elem" type="Regex" minOccurs="1" maxOccurs="unbounded"/></xsd:choice></xsd:complexType>
name="Regex"><xsd:restriction                base="xsd:string"><xsd:pattern
value="."/></xsd:restriction></xsd:simpleType></xsd:schema>

The XSD defines a schema for an XML document.  The XML document can
have multiple elements, each of which is a simple type.  The simple
type is defined as a string that matches the regular expression . The
question is, what is the correct XML document that validates against this XSD?

Pick your answer from the options below:

A. <doc><elem>abc</elem><elem>123</elem></doc>
B. <doc><elem>abc123</elem></doc>
C. <doc><elem>abc</elem><elem>123</elem></doc>
D. <doc><elem>ab1</elem><elem>
```

Figure 2: An example of the "rambling" behavior of the model, where our simplistic post-processing algorithm (drop everything after the rightmost '>' character) breaks. Therefore, we had to implement a more sophisticated post-processor to extract the first valid meaningful XML root tree.

empirical evidence that the model has successfully adapted to the structural constraints of the dataset.

6.2 Qualitative Results

Qualitative inspection of the generated XML files (samples can be found under Section 10 below) reveals a marked increase in structural diversity. In contrast to the zero-shot baseline, which often relied on repetitive or simplistic patterns, the LoRA-tuned model demonstrates a broader utilization of complex XSD features and schema variants, suggesting a deeper acquisition of the underlying syntax.

7 Analysis and Discussion

7.1 Perplexity as a Proxy for Structure

A core question in this study is the relationship between the training metric and the downstream fuzzing goal. We measure **Perplexity** (PPL), defined as the exponentiated cross-entropy loss[8]:

$$PPL(X) = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log p_\theta(x_t|x_{<t})\right) \quad (1)$$

In natural language, low perplexity implies fluency. In our context of structured data, we interpret low perplexity as *structural coherence*. A model with low perplexity on the W3C validation set has effectively learned the "grammar" of XML (e.g., that '<' must be followed by a tag name, and tags must match). This allows us to optimize for syntax validity (passing the parser's gatekeeper) using standard causal language modeling objectives.

7.2 Memorization vs. Generalization

A significant risk in training on repeating, specialized datasets is *verbatim memorization* (Huang et al., 2024). If the model achieves extremely low perplexity (near 1.0), we have a risk that the model will simply regurgitate the W3C training files verbatim.

For fuzzing, a "verbatim memorization" model is not very useful because the target library is already tested against the W3C suite. We require a model that is a bit *confused* (pardon our made-up term): one that mimics the *structure* of valid XML (syntax) but hallucinates the *content* (semantics).

During our experiments, we did encounter final models with perplexity very close to 1. We attempted to mitigate this trade-off by:

1. **Early Stopping:** We monitored validation loss to retain the best performance model right before overfitting kicks in (by setting `eval_strategy` and `load_best_model_at_end`) before the model began to overfit and memorize specific

[8] https://huggingface.co/docs/transformers/v5.0.0rc1/en/perplexity

Experiment Configuration	Validity Rate	Achieved Coverage
Zero-Shot Baseline (Qwen3-0.6B)	0.01	0.08
Fine-Tuned Model (Section 4)	0.15	0.12

Table 1: Model performance comparison between successive experiment configurations.

file contents.

2. **High-Temperature Sampling:** By setting $T = 0.8$, we force the model to sample from the tail of the distribution, ensuring that even if it remembers a W3C example, it introduces sufficient noise to possibly create a novel mutation.

8 Limitations and Ethical Considerations

AI Use Disclosure. We acknowledge the use of artificial intelligence tools (Google Gemini) for editorial review and language refinement during the preparation of this report. All ideas, code, citations, and experimental work presented herein are the original work of the author.

Challenges. Due to the high computational cost of running multiple fuzzing campaigns to establish statistical significance (Böhme et al., 2022), we could not benchmark our approach against external baselines within the project timeline. Furthermore, resource constraints limited our ability to perform a comprehensive hyperparameter search, potentially constraining the final performance and generalization capability of the LoRA-tuned model. Finally, our current training objective *minimizes perplexity* rather than explicitly *maximizing the diversity of generated outputs*. While traditional mutation-based fuzzers achieve diversity through stochasticity, our model's adherence to the training distribution may bias it towards common patterns, potentially delaying the discovery of rare edge cases compared to methods that prioritize input diversity.

Ethical Considerations. We acknowledge the *dual-use nature* of automated vulnerability discovery, which can be employed for both auditing and exploitation. However, our research is grounded in the belief that robust, accessible fuzzing tools are essential for defensive security. They enable maintainers to detect vulnerabilities earlier in the development lifecycle, thereby preempting potential attacks.

9 Conclusion and Future Work

In this work, we investigated the efficacy of adapting Small Language Models for structural fuzzing via Low-Rank Adaptation. Although the fine-tuned `Qwen3-0.6B` model did not yield a statistically significant increase in code coverage relative to the zero-shot baseline, the approach demonstrates fundamental promise. The model successfully transitioned from generating simplistic, repetitive XML structures to producing more complex and diverse XML/XSD outputs capable of engaging the target parser, suggesting that while the current optimization for perplexity may be insufficient for maximizing edge-case discovery, the underlying methodology provides a viable foundation for future, coverage-guided neural fuzzers.

Future Work. While this project showed that structural fuzzing using deep learning is a feasible approach, more research should be done to further increase its efficacy. Future research should move beyond imitation learning by employing Reinforcement Learning (RL), utilizing code coverage metrics as a direct reward signal to optimize the generation policy via Proximal Policy Optimization (PPO). Furthermore, integrating whitebox feedback—such as conditioning the model on real-time coverage maps or source code context—could transform the system from a passive generator into an active, targeted mutator capable of systematically exploring unexecuted code paths. Additionally, the scope could be expanded to generate coupled Schema-Instance pairs, enabling the testing of complex cross-validation logic often missed by single-file generators. Finally, beyond direct data generation, we aim to investigate the capability of language models to synthesize specialized, executable data generators from natural language specifications, effectively automating the creation of custom fuzzing harnesses.

Our takeaways include:

- **Iterative Experimentation:** Finding the right balance between learning the grammar and introducing slight edge case behaviors required significant trial and error. We found that frequent model checkpointing was essential for maintaining experimental velocity, particularly when operating within ephemeral environments like Google Colab where interrup-

tions are not rare.

- **Base vs. Finetuned Models:** We observed that models subjected to heavier instruction tuning (e.g., Q&A fine-tuning) exhibit a tendency towards verbose "rambling" (Figure 2), which interferes with strict syntax generation and degrades performance. We attribute this phenomenon to the model's alignment towards conversational or explanatory outputs (common in exam-based training data) which a raw base model naturally avoids. While utilizing a base model effectively mitigates this verbosity, further investigation is required to determine if doing so compromises the reasoning capabilities necessary for constructing complex, semantically valid structures.
- **Viability of the Approach:** Although resource constraints prevented large-scale statistical validation, the observed improvements in coverage metrics suggest that SLM-based structural fuzzing is a highly promising direction that warrants further research.

10 Reproducibility and Artifacts

The corresponding resources for this report can be found at (please note that Colab notebooks and Google Drive access require a valid NYU account):

Training Notebook:

- Colab: https://colab.research.google.com/drive/1-h3L_-LImhFZgxvf3lDDCG4BNfQ5SBY0?usp=sharing
- GitHub: https://github.com/chitoge/qd2121_dl_final/blob/main/training_notebook.ipynb

Model Weights: https://drive.google.com/drive/folders/1pHanl8OWWV6EttPyYUsOpE9YZTvbJ1Os?usp=drive_link

Baseline Samples: https://drive.google.com/drive/folders/1Lb9TUKNMxBkhltRvpFaODm_hAA66g6k-?usp=drive_link

Fine-Tuned Model Samples: https://drive.google.com/drive/folders/1g7ycUC1xItcVdQ-OfmKNQBLYuLGSWG6Q?usp=drive_link

GitHub Repo: https://github.com/chitoge/qd2121_dl_final

References

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*, ICSE '22, page 1621–1633, New York, NY, USA. Association for Computing Machinery.

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. *Preprint*, arXiv:2212.14834.

Thanh Do. 2025. Midterm project report - team: Spline reticulator. *ECE-GY 7123 / CS-GY 6953 / Deep Learning - Fall '25*.

Alastair F. Donaldson, Cristian Cadar, Manuel Carrasco, Dan Iorga, Daniel Liew, and John Wickerson. 2025. When you have a fuzzer, everything looks like a reachability problem. In *Reachability Problems: 19th International Conference, RP 2025, Madrid, Spain, October 1–3, 2025, Proceedings*, page 3–16, Berlin, Heidelberg. Springer-Verlag.

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learnfuzz: Machine learning for input fuzzing. *Preprint*, arXiv:1701.07232.

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. *Preprint*, arXiv:2106.09685.

Jing Huang, Diyi Yang, and Christopher Potts. 2024. Demystifying verbatim memorization in large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 10711–10732, Miami, Florida, USA. Association for Computational Linguistics.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Qwen Team. 2025. Qwen3: Think deeper, act faster.

Henry S Thompson and 1 others. 2004. Xml schema part 1: Structures second edition. W3C Recommendation.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–13. ACM.