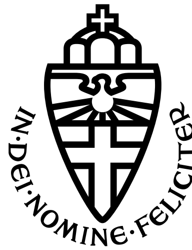
RADBOUD UNIVERSITY



and

Ultraware Consultancy and Development B.V.

Faculty of Science

Utilizing Large Language Models for Fuzzing: A Novel Deep Learning Approach to Seed Generation

November 17, 2023

Master Thesis

Elwin Tamminga (s1013846)

Supervised by:
Bouwko van der Meijs, Ultraware
Stjepan Picek, Radboud University

Abstract

This thesis introduces a novel deep-learning approach to seed generation for fuzzing engines by utilizing a large language model (LLM) based on the Transformer architecture. We developed a user-friendly and highly configurable application that implements this approach, which we published on GitHub.[1] Our implementation is compatible with various LLMs and different state-of-the-art fuzzers, such as AFL++ and libFuzzer. The main advantage of this approach is that it does not depend on a specific fuzzing technique or programming language. Another advantage is that newer models can easily be configured in the application.

An experimental methodology was adopted to demonstrate the effectiveness of our approach in improving the performance of state-of-the-art fuzzing in Go with libFuzzer. Various open-source and closed-source models were fine-tuned for seed generation. These were evaluated and compared against prompt-tuning using our implementation.

A comprehensive analysis of the results shows that the performance of state-of-the-art fuzzing in Go can indeed be improved using LLMs for initial seed generation. This improvement is most notable in the first 30 seconds of fuzzing, where fuzzing with seeds generated by the best-performing models triggers a bug twice as likely compared to fuzzing without seeds. Furthermore, we found that libFuzzer can discover a wider variety of bugs more quickly when fuzzing with seeds generated by LLMs. This resulted in the discovery of several new bugs in the libraries containing the code used during evaluation.

However, the significance of improvement is heavily influenced by several variables including the model used for seed generation, the fuzzing test, and the specifications of the fuzzing machine. Our findings reveal that, in general, the prompt-tuned CodeGen-Multi 16B and prompt-tuned StarCoderPlus models are the best-performing models in increasing fuzzing performance when used for seed generation in Go. This is done most effectively by starting with fuzzing without seeds, utilizing the CPU, while seeds are generated using the GPU.

[1] https://github.com/elwint/seedai

Acknowledgements

I would like to express my deepest gratitude to my university supervisor Stjepan Picek and my company supervisor Bouwko van der Meijs for their time and patience, valuable feedback, and guidance. I would also like to thank Ultraware for supporting this project and providing me with the resources I needed to complete this project, which would not have been possible without their support. Special thanks should also go to my colleagues who supported me during this project. Lastly, I'd like to thank my friends and family members for their moral support.

Contents

1 Introduction **5**

2 Background **7**

2.1 Fuzz testing . . . 7

2.2 The Transformer architecture . . . 8

2.3 Pre-trained LLMs . . . 8

2.3.1 Autoregressive decoder-only models . . . 9

2.3.2 Autoencoding encoder-only models . . . 10

2.3.3 Sequence-to-sequence models . . . 10

3 Related work **12**

3.1 RNN models . . . 12

3.2 GAN models . . . 12

3.3 LLM-based fuzzers . . . 13

4 Methods **14**

4.1 Model selection . . . 14

4.2 Fine-tuning models . . . 15

4.2.1 Dataset . . . 16

4.2.2 Cross entropy loss . . . 18

4.3 Implementation . . . 18

4.3.1 Configuration . . . 20

4.3.2 Fine-tuned models . . . 20

4.3.3 Prompt-tuning base models . . . 20

4.4 Evaluation . . . 21

5 Results **24**

5.1 Fine-tuning . . . 24

5.2 Evaluation and prompt-tuning . . . 26

5.3 Best model configurations and coverage . . . 33

6 Conclusions and Future Work **35**

A Configurations **42**

B Source code evaluation test cases **44**

C Seed generation times **48**

D Other evaluation results **50**

1 Introduction

In recent years, there has been an increasing trend towards using machine learning techniques for vulnerability detection [11, 29]. For instance, we could use the supervised machine learning paradigm and the classification task to classify whether an input, such as the source code of a program, is vulnerable or not [39]. In the field of fuzzing, which is a technique for identifying software bugs (including security bugs) by providing semi-random inputs to a program, machine learning can be used to improve the performance of fuzzing engines to detect software bugs more effectively [71].

There are various ways to apply machine learning in fuzzing [71]. One possible application of machine learning in fuzzing is during seed generation. Fuzzing engines use seeds as initial input to gather a base coverage of the program, which are later mutated by the fuzzing engine to increase coverage to identify software bugs. Seeds can be generated by processing the source code of a program, which can be considered a task similar to processing natural language (NL) [74]. By understanding the context of the source code, including code comments, features can be extracted that can be useful for fuzzing.

State-of-the-art machine learning techniques for natural language processing have recently proven to be effective for a wide variety of tasks, including understanding and generating programming languages (PL) [13, 20]. These techniques utilize a large language model (LLM) such as GPT-4 [53], which is a deep neural network based on the Transformer architecture [65] trained on a large amount of data.

Very recently, LLMs have also emerged in the field of fuzzing. During the course of this thesis project, several papers have been published about utilizing LLMs for fuzzing [17, 18, 75]. These studies mainly take advantage of the LLM's capability to generate code to test compilers and runtime engines, but they do not evaluate the use of LLMs for general-purpose fuzzing with existing state-of-the-art fuzzers.

Other existing deep learning techniques in fuzzing often use a model based on long short-term memory (LSTM) [71]. However, LLMs based on the Transformer architecture have already demonstrated superior performance compared to LSTM models when it comes to processing source code for a wide variety of tasks [20]. Another problem with existing techniques is that they are not user-friendly or only implemented for a specific fuzzer. Furthermore, many existing techniques are designed for a specific programming language and cannot easily be applied to another programming language.

To address these challenges, we introduce a novel deep-learning approach to seed generation for fuzzing engines using a state-of-the-art LLM based on the Transformer architecture. The model processes the source code of a fuzzing test, including the underlying functions it calls, to generate initial seed files. One advantage of this approach is that it does not depend on a specific fuzzing technique or programming language. It also has an advantage over a vulnerability prediction model because a fuzzer can prove if the code is vulnerable or not by exploiting the bug using the generated seeds (by failing the fuzzing test).

Our approach requires a pre-trained state-of-the-art LLM that has been trained on data containing the same programming language as the programs being tested, and natural language (English) since source code may include English-written comments that can help the model to understand the context. The Go programming language is the main programming language

used within Ultraware, where this master thesis project was done. Therefore, we only focused on this programming language during our research and evaluation. However, our approach should also work with other LLMs trained or fine-tuned for other programming languages. Since there are many different types of Transformer-based LLMs, we formulated the following research questions for our approach:

RQ1: What are the best performing LLMs capable of understanding Go code?
RQ2: How can LLMs be implemented for seed generation to improve the performance of state-of-the-art fuzzing in Go most effectively?
RQ3: Can fine-tuning LLMs for seed generation further improve fuzzing performance?

The next section describes background information and existing literature that was analyzed for RQ1. The related work section describes existing deep-learning techniques for seed generation in fuzzing. Different fuzzing experiments were conducted to evaluate the best-performing models from the literature to answer RQ2. These were also fine-tuned and evaluated to answer RQ3.

Our results show that the performance of state-of-the-art fuzzing in Go can be improved using our approach. We also evaluated and compared our implementation with different configurations and baselines. Altogether, we made the following main contributions:

- To the best of our knowledge, we are the first to fine-tune, implement, and evaluate various LLMs for initial seed generation compatible with existing state-of-the-art fuzzers.
- We developed a user-friendly and highly configurable implementation of our approach that can be re-used with another model, programming language, or fuzzer.
- Using our approach, we discovered several new bugs in the libraries containing the code used for creating the test cases during evaluation.
- We provide the datasets used during fine-tuning and the source code of our implementation.

The scripts used for fine-tuning the models have been published on GitHub.[1] The created training and validation datasets used for fine-tuning have been uploaded to HuggingFace.[2] The source code of our implementation has also been published separately on GitHub,[3] as well as the raw evaluation results, fuzzing logs, generated seeds, and scripts used for evaluation.[4]

---

[1]`https://github.com/elwint/thesis-fuzz-finetune-scripts`
[2]`https://huggingface.co/elwint#datasets`
[3]`https://github.com/elwint/seedai`
[4]`https://github.com/elwint/thesis-seedai-evaluation`

2 Background

2.1 Fuzz testing

Fuzz testing, also known as fuzzing, is a technique for discovering bugs in a program by providing semi-random inputs to a program [24]. Through the use of fuzzing, a large number of security vulnerabilities in various programs have been discovered, which made it increasingly more popular among security researchers [35].

Fuzzing starts with defining unexpected behavior of the program under test that should be counted as a bug. This is often done by creating a fuzzing test, which is similar to a unit test but without defining predetermined inputs [62]. A fuzzer then tries to fail the fuzzing test by providing inputs. There are various forms of fuzzing [38], with gray-box fuzzing being the most common form of fuzzing [35]. Gray box fuzzing strikes a balance between white-box fuzzing (full knowledge of the internal structure) and black-box fuzzing (no knowledge of the internal structure) [38].

State-of-the-art fuzzers, such as AFL++ [63] and libFuzzer [40], are examples of coverage-based gray-box fuzzers. These fuzzers use a process of mutating inputs based on the feedback received from code coverage analysis [35, 38]. They start with initial seed inputs, which are basic examples of valid data expected by the program. The fuzzer then modifies these seeds in various ways to generate a wide range of inputs. By monitoring which parts of the code are executed with each input, the fuzzer can refine its future mutations to explore more paths in the program's code. This method is particularly effective since it allows the fuzzer to navigate through the program systematically, increasing the chances of discovering bugs. An overview of the coverage-based fuzzing process is illustrated in Figure 1.

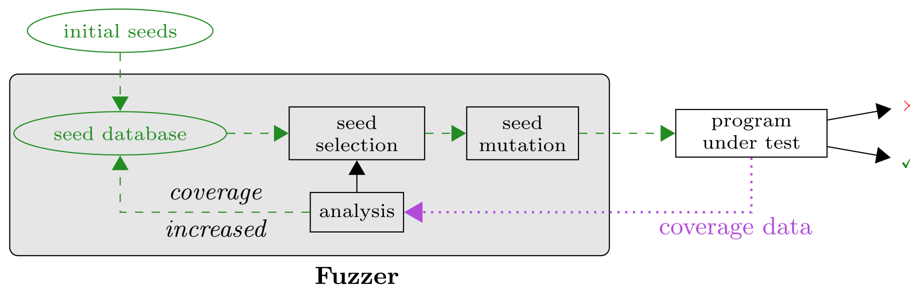


Figure 1: Coverage-based fuzzing process [66]

Fuzzing is predominantly utilized in programming languages that lack certain safety guarantees, such as C/C++. However, fuzzing is equally applicable to safer languages, such as Go and Rust [60]. This has already been demonstrated by the vast amount of bugs found by Go-specific fuzzers [67]. One of the leading fuzzers compatible with Go is libFuzzer [77], a popular state-of-the-art fuzzer. It uses state-of-the-art fuzzing techniques such as the NeverZero strategy [77] and Entropic [5], which are currently leading in fuzzing performance based on recent benchmarks [25]. Additionally, libFuzzer can reject unwanted inputs and supports value profiling for smarter coverage guidance. Coverage instrumentation for libFuzzer can be done natively using the Go compiler [77].

Another recent development in the field of fuzzing is the application of machine learning techniques to enhance fuzzing effectiveness. These techniques can be integrated at various stages, such as initial seed generation, seed selection, seed/input mutation, or exploitability anal-

ysis [71]. Our research has specifically focused on initial seed generation using large language models to address the challenges outlined in the introduction. Other existing machine-learning techniques for seed generation are described in the related work section.

2.2 The Transformer architecture

The Transformer, introduced in the paper "Attention Is All You Need" [65], is a neural network architecture that has shown superior performance in a wide variety of natural language processing (NLP) tasks [53]. The Transformer model was introduced as a replacement for traditional recurrent neural network (RNN) models, such as long short-term memory (LSTM), for language modeling and translation tasks. One key advantage of the Transformer architecture is its ability to efficiently handle long variable-length input sequences. Unlike RNN models, which rely on sequential processing and can be slow and difficult to parallelize, the Transformer can process input sequences in parallel, making it much more efficient for long input sequences.

The original Transformer is an encoder-decoder architecture, composed of an encoder and a decoder segment. The encoder generates a high-dimensional representation of the input data, which is used as input to the decoder. The decoder uses this input together with the current output to generate new sequences. Together these components can create a sequence-to-sequence (seq2seq) model [23]. Both the encoder and decoder rely on a self-attention mechanism, which allows the model to focus on different parts of the input sequence at different times during the computation. This self-attention mechanism allows the model to encode contextual information from the input sequence to generate a high-dimensional representation.

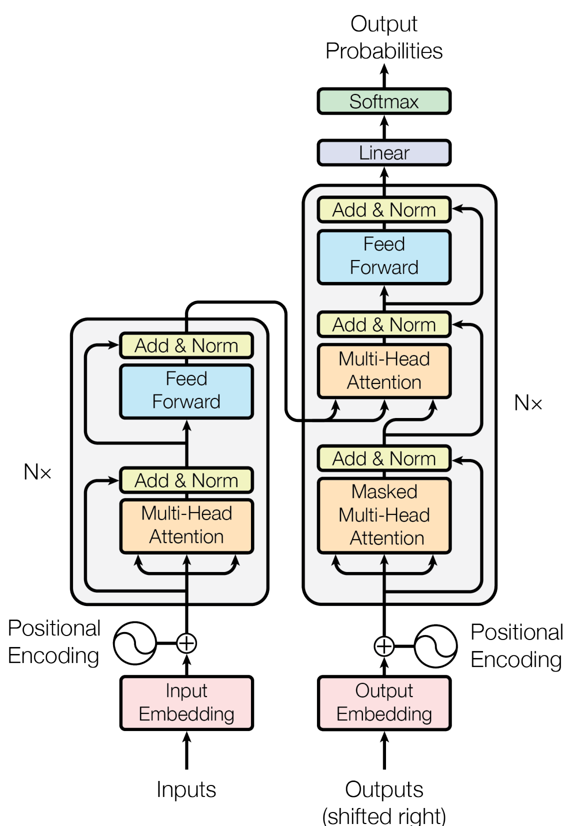


Figure 2: The Transformer architecture [65]

The core of the Transformer architecture is composed of a series of layers consisting of positional encoding, multi-head self-attention modules, and a feed-forward network. Using positional encoding, the model can determine the position of each word and its distance to other words. The multi-head self-attention mechanism is designed to simultaneously focus on and process different segments of the input sequence.

2.3 Pre-trained LLMs

Large language models (LLMs) have become very popular in natural language (NL) and programming language (PL) processing tasks since the introduction of the Transformer architecture [7]. Most pre-trained state-of-the-art LLMs use either the encoder, decoder, or both compo-

nents of the original Transformer architecture for different purposes [57]. These models can be divided into three categories: autoregressive decoder-only, autoencoding encoder-only, and sequence-to-sequence (seq2seq). We mainly focus on LLMs trained on data containing English and the Go Programming Language, which can be used for our approach.

2.3.1 Autoregressive decoder-only models

Autoregressive models are trained on the classic language modeling task: given a sequence of tokens, predict the next token [55]. They often correspond to the decoder part of the Transformer architecture, and use an attention mask to prevent the model from seeing future tokens. This is also called causal language modeling or causal masking [56].

Most Transformer-based LLMs that are capable of understanding and generating code are autoregressive decoder-only models [9]. A common method for evaluating autoregressive coding models is the HumanEval benchmark created by OpenAI, which evaluates a model's performance on its Python code-writing capabilities using a context consisting of code and/or comments written in English [13]. However, depending on the training data distribution of the model, this may not give a good indication of its performance on Go code since it could have mainly been trained on Python code.

Recognizing this limitation, the HumanEval benchmark has been adopted into multi-language benchmarks by other researchers to provide a more comprehensive evaluation. At the time of writing, there are three Go-variants of the HumanEval benchmark: MBGP [2], HumanEval-X [79], and MultiPL-E [10]. To the best of our knowledge, no evaluations have been done using the MBGP benchmark.

The following autoregressive models are trained on datasets containing Go code. The table presents the pass@k score, which is an unbiased estimator most commonly used to compare HumanEval-evaluated models [13].

Table 1: Pass@1/100 scores of decoder-only models capable of understanding Go code

Model	Size	HE @1	HE @100	HE-X Go @1	HE-X Go @100	MultiPL-HE Go @1
GPT-4 [53] *	?	67.0%	-	-	-	-
GPT-3.5 [53] *	? **	48.1%	-	-	-	-
StarCoderBase [36]	15.5B	30.4%	-	-	-	21.47%
LLaMA [64]	65B	23.7%	79.3%	-	-	-
PaLM [16]	540B	26.2%	76.2%	-	-	-
MIM-2.7B [46] †	2.7B	30.7%	69.6%	17.4%	53.6%	-
CodeGeeX [79]	13B	22.89%	60.92%	14.43%	47.14%	11.04%
replit-code-v1 [58] ‡	3B	17.0-21.9%	-	-	-	-
BLOOM [59]	176B	15.52%	55.45%	-	-	-
CodeGen-Multi [48]	16B	18.32%	50.80%	13.03%	48.77%	13.54%
InCoder [21]	6.7B	15.2 %	47.0%	8.68%	28.31%	< 9%
CodeGen-Multi [48]	6.1B	18.16%	44.85%	9.98%	41.01%	-
GPT-NeoX [3]	20B	15.4%	41.2%	5.00%	32.08%	-
LLaMA [64]	7B	10.5%	36.5%	-	-	-
BLOOM [59]	7.1B	7.73%	29.47%	-	-	-
GPT-J [68]	6B	11.62%	27.74%	4.01%	23.70%	-
PolyCoder [76]	2.7B	5.59%	17.68%	-	-	-

The scores in Table 1 were found in the corresponding model paper or other previous research. Scores with additional generation strategies (e.g. prompt fine-tuning [69], CodeT [12], Reflexion [61]), instruction-tuned models (e.g. WizardCoder [42]), and models specifically trained/fine-tuned for Python (e.g. CodeGen-Mono [48], StarCoder [36]) have been excluded for a fairer comparison.

OpenAI's GPT models perform the best on the HumanEval benchmark. However, these models are not publicly available, and can only be accessed through OpenAI's API. Furthermore, the authors of PolyCoder, who conducted a systematic evaluation on various models [76], mention that "since the exact training set of Codex", which is a descendant of GPT-3 [51], "is unknown, it may include files from these test sets rendering Codex's results overly-optimistic." The best-performing open-source decoder-only model is currently StarCoderBase.

2.3.2 Autoencoding encoder-only models

Autoencoding models are trained by corrupting the input tokens in some way and trying to reconstruct the original sequence [55]. They correspond to the encoder part of the original Transformer model and have access to the full input sequence without any mask. This means that autoencoding models usually build a bidirectional general-purpose representation of the whole sequence, which can easily be fine-tuned for tasks such as sentence classification or code search.

A popular autoencoding model is BERT [19]. However, BERT's training data does not contain any source code. CodeBERT is a 125M autoencoding model that is specifically trained for understanding programming languages [20]. It performs well on Go with code search and predicting masked code tokens on CodeXGLUE, a benchmark dataset for code understanding and generation [41]. A comparison of the accuracy scores of CodeBERT between Go and the average of all tested languages is presented in Table 2.

Table 2: CodeBERT Go accuracy scores comparison

Task	Go	All
NL code retrieval (CodeSearchNet)	88.2%[72]-84% [20]	69.3%[72]-76%[20]
Masked code token prediction (PL-probing)	90.79%[20]	85.66%[20]

Although CodeBERT performs well on Go, newer sequence-to-sequence models such as CodeT5+ already surpass CodeBERT in various encoding benchmarks, including CodeXGLUE [72]. For instance, the CodeT5+ 770M model has a score of 92.7% on natural language code retrieval on CodeSearchNet on Go code [72].

2.3.3 Sequence-to-sequence models

A sequence-to-sequence (seq2seq) model combines the elements of autoencoding and autoregressive modeling into one model. An example of a seq2seq model is the original Transformer [55], which consists of an encoder and decoder segment. This type of model is commonly used

*The training datasets of OpenAI models are unknown.

**GPT-3 contains 175B parameters [8], but the exact size of GPT-3.5 is unknown.

†The MIM-2.7B model is not yet publicly available, scores are not verified.

‡No paper exists for the replit-code-v1-3b model, scores may be overly-optimistic.

for translation tasks because it transduces the whole input sequence into a new sequence. A popular seq2seq model is T5 [56], which can translate text into another language.

Using the architecture of T5, CodeT5 was trained on various programming languages and is one of the first models capable of executing various code-related tasks, such as NL-PL (from natural language to code), PL-NL (from code to natural language) and PL-PL (source code translation). However, after CodeT5, more autoregressive models emerged that are also capable of executing these tasks with better performance by only using a left-to-right autoregressive decoder [79].

Another seq2seq architecture is the Prefix-LM architecture [56]. This architecture is used in a coding model called CodeGen2 [49]. In contrast to a distinct encoder and decoder segment, the Prefix-LM architecture combines an encoder with bi-directional attention and an autoregressive decoder with causal masked attention into the same segment. However, the authors of CodeGen2 concluded that the Prefix-LM architecture does not provide any benefits over the decoder-only baseline on both code generation and infilling.

Despite the contrary findings of the CodeGen2 model, a newer variant of the CodeT5 model was developed. This variant is called CodeT5+ [72]. It not only performs better than other seq2seq models, but it also surpasses many other decoder-only models, including CodeGeeX and StarCoder, in the HumanEval benchmark [72]. Table 3 presents the HumanEval benchmark results of the seq2seq models.

Table 3: HumanEval scores of CodeGen2 [49] and CodeT5+ [72]

Model	**Size**	**Pass@1**	**Pass@10**	**Pass@100**
InstructCodeT5+ *	16B	35.0%	54.5%	77.9%
CodeT5+	16B	30.9%	51.6%	76.7%
CodeGen2.5-Multi	7B	28.36%	47.46%	75.15%
CodeT5+	6B	28.0%	47.2%	69.8%
CodeT5+	2B	24.2%	38.2%	57.8%
CodeGen2.0	16B	20.46%	36.50%	56.71%
CodeGen2.0	7B	18.83%	31.78%	50.41%
CodeT5+	770M	15.5%	27.2%	42.7%
CodeT5+	220M	12.0%	20.7%	31.6%

The CodeT5+ paper notes that both CodeT5+ 220M and 770M utilize the same T5 architecture and were trained from scratch. However, the larger CodeT5+ models adopt a different "shallow encoder and deep decoder" architecture [72]. For these models, encoders are initialized from CodeGen-mono 350M, whereas decoders are initialized from CodeGen-mono of the corresponding size.

*InstructCodeT5+ is the instruction-tuned variant of CodeT5+ (fine-tuned to respond to instructions) [72].

3 Related work

Utilizing LLMs for fuzzing is a development that has emerged very recently. At the start of this thesis project, only a few techniques existed that use deep learning (DL) for seed generation [71]. Most of these techniques use a recurrent neural network (RNN), but some techniques also use a generative adversarial network (GAN). This section explores the most relevant studies about these older and more recent techniques.

3.1 RNN models

One study developed a fuzzer called NeuFuzz that uses deep learning for seed generation [73]. NeuFuzz is based on a binary grey-box fuzzer called PTfuzz, which is guided by program traces collected during execution [78]. NeuFuzz utilizes these traces using a long short-term memory (LSTM) model to prioritize seeds that can trigger certain paths that are predicted to be vulnerable. This is done by transforming program execution paths into vector representations using one-hot and word2vec, which are word embedding methods commonly used in language processing tasks. The paper argues that NeuFuzz can discover more vulnerabilities than the existing fuzzers in less time although the significance of this improvement is not mentioned.

The authors of NeuFuzz explored existing datasets to train their model. They found that the dataset provided by the authors of VulDeePecker, a DL-based system for vulnerability detection [37], is not an adequate training dataset. The paper mentions that “VulDeePecker conservatively believes that code is non-vulnerable as long as no vulnerability has been found in them; in reality, this assumption does not hold.” Therefore, the authors of NeuFuzz created their own binary dataset to train NeuFuzz’s neural network, in which they only assume that the code is non-vulnerable after patching. This dataset is derived from three sources: the NIST SARD project, GitHub, and Exploit-DB. The binaries collected from these sources were selected based on common vulnerabilities in the C programming language. It includes both pre-patch and post-patch versions and test cases for program execution. In total, the training and testing of the neural network was carried out on a total of 28475 vulnerable binaries and 27436 non-vulnerable binaries. However, the compiled dataset has not been made public.

Another study developed a machine learning-based framework trained on execution paths to improve the quality of PDF seed inputs for fuzzing programs [15]. Similarly to NeuFuzz, the framework uses a recurrent neural network. They reported significant increases in code coverage and the likelihood of detecting program crashes, demonstrating the potential of deep learning techniques for seed generation in fuzzing. Unfortunately, the details regarding their specific architecture and code were not provided.

3.2 GAN models

SmartSeed is a seed generation system that uses a GAN model [43]. Unlike previous models that primarily utilize RNNs, the use of a GAN may allow the generation of synthetic data that is more similar to real seed data. The workflow of SmartSeed consists of three stages: preparation, model construction, and fuzzing. In the preparation stage, inputs are collected from a fuzzing tool directly, which is then used to train the GAN model in the model construction stage.

The results from the study highlighted several advantages of SmartSeed. First, it requires only tens of seconds to generate sufficient high-quality seeds. Second, it can generate seeds for

various input formats. Third, SmartSeed is compatible with different fuzzing tools. In their tests, SmartSeed discovered more than twice as many unique crashes and 5040 extra unique paths compared to the existing best seed selection strategy for the evaluated 12 applications. Despite these promising results, the code for SmartSeed has not been made public.

It remains unclear whether the GAN model used in SmartSeed outperforms RNNs for seed generation in fuzzing. Another paper argues that a GAN model outperforms an LSTM model when it is used for reinitializing AFL with novel seed files [47]. Their implementation helped AFL discover over 14% more code paths compared to a random strategy. However, like SmartSeed, the code has not been made public, limiting external validation and further research.

3.3 LLM-based fuzzers

Very recent related studies have also started to explore the use of LLMs in fuzzing. Fuzz4All is a standalone fuzzer leveraging LLMs as an input generation and mutation engine to produce diverse and realistic inputs for the target programming language [75]. They use the GPT-4 [53] model to automatically generate prompts and the StarCoder [36] model to generate fuzzing inputs during the fuzzing loop. They identified 76 bugs in mainly compilers and runtime engines, such as the Go toolchain, GCC, Clang, and OpenJDK.

The fuzzer utilizes the LLMs to generate code as fuzzing inputs, which is then compiled and executed to discover bugs in the tested components. The study describes that state-of-the-art fuzzers, such as libFuzzer and AFL++, are general-purpose fuzzers. These struggle to produce programming language fuzzing inputs [75], which are important for fuzzing compilers and runtime engines.

For Go, the authors of Fuzz4All compared their results with go-fuzz, a general-purpose coverage-based fuzzer for Go programs heavily based on AFL [67]. However, they did not compare their results with state-of-the-art fuzzers such as libFuzzer, which is also compatible with Go. Furthermore, go-fuzz is effectively deprecated according to the maintainer since the introduction of the builtin Go fuzzer.*

A similar LLM-based fuzzer is TitanFuzz, which is a fuzzer for fuzzing deep learning libraries [18]. It utilizes the LLMs to generate Python code to discover bugs in the libraries. A potential bug is detected through differential testing between the CPU and GPU during execution. Its successor is FuzzGPT, substantially outperforming TitanFuzz. FuzzGPT synthesizes unusual programs for fuzzing and includes a fine-tuned LLM [17].

Our approach is different from the studies above since we focus on general-purpose fuzzing with existing state-of-the-art fuzzers compatibility. The capability of an LLM to understand code and generate bug inputs is more important for our approach than code generation. Furthermore, we analyze more LLMs, including fine-tuning, using comprehensive evaluation metrics.

Finally, the preliminary report of another study indicates that the effectiveness of a basic mutation fuzzer can be improved with an LLM that generates seeds based on a natural language format specification [1]. The generated seeds are well-formed to reach various parts in the given specification. Unfortunately, the authors of this study have not yet evaluated their method with state-of-the-art fuzzers, and the results have not yet been published.

---

*`https://github.com/dvyukov/go-fuzz/issues/329`

4 Methods

We adopted an experimental methodology to demonstrate the effectiveness of our approach in improving the performance of state-of-the-art fuzzing in Go using various models. Our approach requires a pre-trained LLM since these can be fine-tuned or prompt-tuned for downstream tasks. This saves time and resources compared to training a new model from scratch, which requires large amounts of training data. We only focused on Transformer-based LLMs, because the Transformer architecture has shown promising results on various NL/PL processing tasks [7, 53]. We utilized the models to process the source code of a fuzzing test and the underlying functions it calls to generate high-quality fuzzing inputs as initial seed files compatible with state-of-the-art fuzzers. Our methodology is split into four parts: the selection of the models, fine-tuning the models, implementing the models, and the evaluation of the effectiveness of our implementation for each model.

4.1 Model selection

The background section describes different large language models that are capable of understanding and generating Go code and the English language. Understanding natural language is a necessary requirement since source code may include English-written comments that can help the model understand the context. The benchmarks used to measure the performance of the models in the literature may not necessarily reflect the model's capability of generating fuzzing inputs. However, given the considerable amount of time required to evaluate all the large language models identified in the literature, we primarily focused on the models that have achieved the highest scores on the relevant benchmarks. We also prioritized smaller well-performing models, because they require minimal resources and can generate fuzzing inputs faster.

Both autoregressive decoder-only models and sequence-to-sequence models were selected for our experiments. They can both utilize the Transformer's ability to capture the entire context of source code along with embedded natural language comments. This is done by the encoder segment in seq2seq models, or the decoder in autoregressive models. The decoder segment in both model types produces the fuzzing inputs used as initial seed files for a fuzzer.

Autoencoding encoder-only models such as CodeBERT were not selected for our experiments. The main reason is its inability to generate fuzzing inputs since it does not incorporate a decoder in its architecture. CodeBERT can be fine-tuned for seed generation by adding a decoder segment, essentially turning it into a seq2seq model. However, since newer seq2seq models such as CodeT5+ already outperform CodeBERT on encoding tasks, we believed that investing time in modifying and fine-tuning encoder-only models would not be worthwhile.

A distinction was also made between closed-source and open-source models. Considering our user-friendly implementation, we had to take into account not only the performance and capabilities of the models but also their accessibility. Closed-source models such as GPT-4 are only accessible through a paid API. Furthermore, the current best-performing open-source models are much smaller than the best-performing closed-source models. Therefore, we experimented with both closed-source and open-source models.

The following LLMs were selected for our experiments: CodeT5+, CodeGen2.5, StarCoder(Base), CodeGen-Multi, GPT-3.5/base, and GPT-4. These models were selected based on accessibility and fine-tuneability, and they scored well on the relevant benchmarks found in previous studies.

We did not select models requiring special access, such as LLaMA, or models with unverified scores. We also did not select very large models that cannot be trained on a single A100 GPU.

4.2 Fine-tuning models

The selected large language models (LLMs) were fine-tuned and/or prompt-tuned for seed generation to use them in our implementation. This section describes the used fine-tuning methods on models that could be fine-tuned and a detailed description of the datasets used for fine-tuning.

Fine-tuning involved retraining the model to focus solely on generating seeds (fuzzing inputs). This method came with several challenges. It requires a dataset containing source code, labeled with an input that causes a bug in the corresponding code. Additionally, training an LLM requires a significant amount of computational resources.

We examined different datasets that were used in past studies [71]. These datasets include collections of test programs with examples of bug-triggering inputs (e.g. NIST SARD and VuldeePecker), as well as datasets specifically designed for vulnerability detection (e.g. VDiscovery), and those intended for benchmarking fuzzers (e.g. LAVA-M). However, the inconsistent labeling of examples in these datasets often made it difficult to extract bug-triggering inputs. In some cases, the code of a program was only labeled as vulnerable or not without further detail. Additionally, to the best of our knowledge, none of these datasets include Go code, thereby limiting fine-tuning performance for Go. Given these limitations, we created a new dataset that includes Go code. The structure of the dataset is described in the Dataset section.

The next challenge was obtaining the necessary computational resources to fine-tune the open-source LLMs. This proved to be more challenging than expected because training an LLM requires significantly more resources compared to merely executing the model. We used a single NVIDIA A100 GPU (80GB) for fine-tuning most models. Some smaller models were fine-tuned using an NVIDIA L4 GPU (24GB). We also had access to a TPU v3 device through Google's TPU Research Cloud (TRC) program [28], but we were not able to use it since many fine-tuning tooling and libraries require NVIDIA's CUDA Toolkit.

Fine-tuning the open-source models was done using the Transformers library from HuggingFace in PyTorch [32]. The primary reason for selecting this library is that it provides an intuitive interface for executing and fine-tuning supported models. Furthermore, because we selected the models based on fine-tuneability and accessibility, the selected open-source LLMs are all publicly available on HuggingFace and compatible with this library. This also simplified the process of developing a user-friendly implementation.

To fine-tune very large language models on a single A100 GPU, we enabled multiple memory-saving techniques available within the Transformers library. This included automatic mixed precision [34, 45] using FP16 (float16) or BF16 (bfloat16 [70]), based on model compatibility. Additionally, we used gradient checkpointing [14]. Some models required small modifications to their code to enable these techniques. Additionally, we utilized Parameter-Efficient Fine-Tuning (PEFT) [44] using Low-Rank Adaptation (LoRA) [31]. Each model's LoRA configuration was adjusted to align with recommendations from the model developers. Since this may affect fine-tuning performance, we also fine-tuned models without LoRA when possible. Finally, we also tested 8-bit configurations, but this did not lead to convergence on any model.

We also experimented with various hyperparameters, but both the batch size and context

length remained constrained by GPU memory limits and the model's maximum context length. Therefore, most models were fine-tuned with a batch size of 1. Nonetheless, given that our dataset was small and that smaller batch sizes tend to work better for smaller datasets [52], we only tested different batch sizes on a smaller model since we believed it would not significantly affect our results. Instead, we optimized other parameters to achieve the best results with the available resources.

Separate scripts were developed for fine-tuning open-source seq2seq and autoregressive models, the primary distinction being their data preprocessing functions. For fine-tuning autoregressive models, we defined which part of the input data corresponds to the completion part since the whole sequence is passed to the decoder. This is done by the preprocessing function, which marks only the completion part (i.e. the expected fuzzing input) for loss calculation. For fine-tuning seq2seq models, this step is not necessary since the architecture contains a separate encoder and decoder segment. The compilation part is simply labeled as output for the decoder, and the source code is labeled as input for the encoder. The rest is then handled by the Transformers library for loss calculation. Fine-tuning was stopped upon the detection of overtraining, signaled by a consistent rise in validation loss.

For the closed-source models, only the GPT-3.5 Turbo and GPT base models could be fine-tuned at the time of our experimentation. This required implementing OpenAI's fine-tuning API and limiting the context length to 4096 tokens [52]. While several base models were available for fine-tuning, they are not mentioned in the GPT-3 [8] or GPT-4 [53] papers. The API documentation mentions that the base models are not trained with instruction following and that the most capable base model is called davinci [54]. The exact differences between these variants remain unclear.

The davinci base model was fine-tuned using the same training and validation dataset as the open-source models. However, when it came to fine-tuning the GPT-3.5 Turbo model, some adjustments were necessary to conform to OpenAI's chat completion API, which requires system, user, and assistant messages. Two versions were fine-tuned. In both versions, the system message was set to "Generate a bug input." In the first version, the system message included the source code from the dataset, and the user message contained the expected fuzzing input. In the second version, the expected input was encoded as a JSON string and prefixed with "A bug input: ". This follows OpenAI's recommendation of embedding instructions within the prompt of each training example that works best for the model prior to fine-tuning [52]. The same prompt must also be used during interference. This was taken into account during implementation.

4.2.1 Dataset

For the creation of our training dataset, we mainly focused on security-related bugs, considering their higher criticality compared to typical software bugs. Furthermore, vulnerable examples could easily be collected from the Go vulnerability database [27]. This database, maintained by the Go team, includes information on vulnerable Go packages or programs. By utilizing the database's API, we created a script to collect vulnerable Go functions, including their underlying functions. It was often necessary to manually construct the corresponding input that could trigger the bug since only a bug description is provided. This turned out to be an intensive task, where a unique input had to be crafted for each unique vulnerability. We partially automated this process using libFuzzer by creating a basic fuzzing test for each vulnerability. As a result, our

training dataset consisted of 193 vulnerable examples representing only 52 unique vulnerabilities. We also created a separate unique training dataset containing only 52 vulnerable examples representing each unique vulnerability.

However, during fine-tuning, we observed that using the whole training dataset resulted in worse performance across all models and various configurations compared to using the unique dataset. We believe that this is a result of using a small unbalanced dataset. The dataset contains a different number of examples for each vulnerability, causing a model to quickly overfit on vulnerabilities with the most examples. Therefore, we decided to only use the unique dataset.

For the creation of our validation dataset, we used a different approach. Assembling a well-structured validation dataset posed certain challenges. As highlighted by the authors of PolyCoder [76], there is a risk that benchmark scores are overly optimistic because the dataset contains data that the model has already seen before. To mitigate this risk, we manually crafted our validation set by randomly selecting distinct packages from GitHub, and we manually introduced a unique bug or vulnerability to each example. This approach ensured that the model did not encounter training data during validation. However, manually creating bugs and inputs was an even more intensive task. Therefore, the validation dataset consists of only 10 examples.

The structure of the datasets follows OpenAI's best practices for conditional generation fine-tuning of the GPT base models [52]. Each example in the dataset contains source code with a weakness and an input that exploits the weakness in the corresponding code. This is embedded into two distinct features, namely, 'prompt' and 'completion'. Both of these features are strings.

The 'prompt' feature contains the source code with a specific weakness followed by a split token (`\n\n###\n\n`). The source code is structured as depth-first package order to allow for truncation. This is necessary because it contains the public vulnerable function and its underlying functions, which together may exceed the model's maximum context length. The pseudocode below describes how the source code is extracted and formatted.

Algorithm 1: Source code parsing

```
Require: f is the function name
 1: function ParseSourceCode(f)
 2:     P ← Process(∅, f, 0)                  ▷ Recursively process the underlying functions
 3:     r ← empty string
 4:     n ← Len(P)
 5:     for i ← 1 to n do                     ▷ Iterate through the packages in processed order
 6:         if i > 1 then
 7:             r ← r + GetPackageName(P_i) + "\n"
 8:         end if
 9:         r ← r + "```" + P_i^r + "```"     ▷ Encapsulate code in code blocks
10:         if i < n then
11:             r ← r + "\n\n"
12:         end if
13:     end for
14:     return r                              ▷ Return the combined source code string
15: end function
```

```
16: function PROCESS(P, f, d)
17:     if d = 5 then                                  ▷ Process up to a depth of 5
18:         return P
19:     end if
20:     p ← GETPACKAGE(f)
21:     if p ∉ P then
22:         P ← P ∪ {p}
23:         p^r ← empty string
24:     end if
25:     p^r ← p^r + GETCODESEGMENT(f)                  ▷ This includes code comments
26:     F ← GETFUNCTIONS(f)
27:     for i ← 1 to LEN(F) do
28:         P ← PROCESS(P, F_i, d + 1)
29:     end for
30:     return P
31: end function
```

The split token is only required for autoregressive LLMs to identify the point of transition from 'prompt' to 'completion'. Thus, the split token was preserved when the source code needed to be truncated. The split token was only removed during fine-tuning the seq2seq models.

The 'completion' feature is the corresponding input that exploits the weakness in the 'prompt' source code, followed by an end token (beginning with a space): `" END"`. During fine-tuning, the end token was replaced by the model's tokenizer end token that was used during pre-training.

4.2.2 Cross entropy loss

Although the selected models differ in architecture, they all used cross-entropy loss during training and fine-tuning. For the open-source models, the cross entropy loss was automatically calculated by the HuggingFace's Transformers library using PyTorch. The library computes a multi-class log loss of the model output logits x_i of size C (the tokenizer vocabulary size) and the corresponding target label y_i for each example i in a batch of size B. The prompt segment of the input along with any special tokens, such as pad tokens, are removed before the loss calculation. The loss formula in the PyTorch library documentation[†] clarifies that a softmax function is used in the loss calculation to normalize the logits x_i. This normalization produces the predicted probabilities $\hat{y}_i = Softmax(x_i)$. The batch loss l_n and the total cost function L for a dataset of N batches are then computed with mean reduction as follows:

$$l_n = -\frac{\sum_{i=1}^{B} \log(\hat{y}_i) \cdot y_i}{B} \tag{1}$$

$$L = \frac{\sum_{n=1}^{N} l_n}{N} \tag{2}$$

4.3 Implementation

We implemented our approach while also addressing issues present in previous techniques mentioned in the introduction. One of these issues is that the implementation cannot be re-used because it requires specific knowledge of its implementation, or it only works with a

[†] `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`

specific fuzzer. Furthermore, many existing techniques are designed for a specific programming language and cannot easily be applied to another programming language. Therefore, we implemented our approach with the following requirements:

1. Other programming languages should be supported.
2. Other fuzzers should be supported.
3. No specific knowledge of our approach should be required to generate initial seed files, i.e. anyone who can run a fuzzer should be able to generate initial seed files without requiring machine learning knowledge.

Our implementation consists of two main components. The main component is a Python program that executes a configurable LLM for inference. It uses the Transformers library from HuggingFace in PyTorch [32] because the selected open-source LLMs are all publicly available on HuggingFace and compatible with this library. The second component is a source code parser, which is executed by the Python program. The parser returns the source code of the provided fuzzing test and the underlying functions it calls. We implemented a Go source code parser in Go, but the parser can easily be replaced to support other programming languages. The data processor in the Python program then processes the source code data so that it can be used as input for the LLM. Finally, the Python program executes the LLM for inference to generate the seed files. Figure 3 presents an overview flow of our implementation.

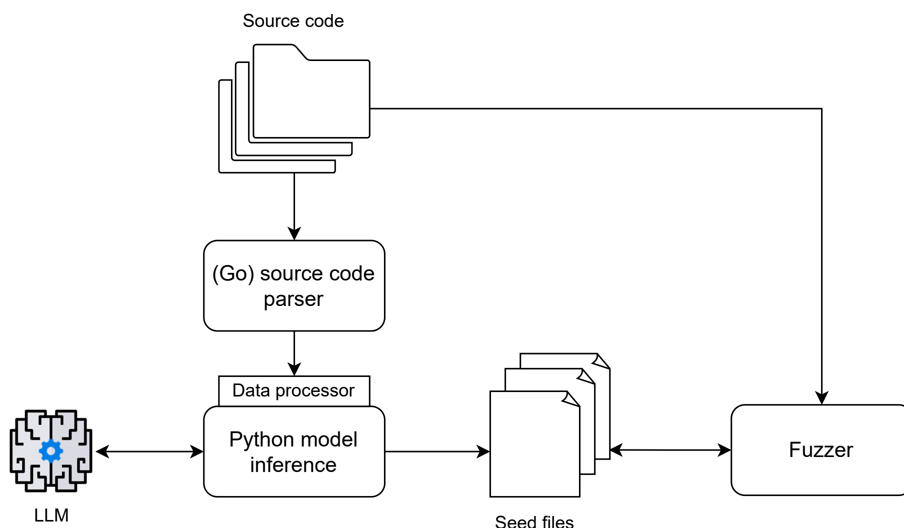


Figure 3: Implementation overview flow

The output of the source code parser should be structured to allow for truncation. When using a fine-tuned model, it should also have the same format as the dataset used during fine-tuning. Additionally, the source code parser should be configurable to return only source code (i.e. without encapsulating the code in code blocks). This is required for code-only base models pre-trained on only source code. For our experiments, we used the same source code parser that was used during the creation of our fine-tuning dataset described in Algorithm 1, with an additional option to remove the encapsulation of the code and adding the package name as code comment.

For decoder-only models, the data processor in the Python program truncates the source code to the maximum context length of the model minus a configurable seed generation length.

By default, a quarter of the maximum context length is reserved for generation. For seq2seq models, the source code is simply truncated to the maximum context length of the encoder.

The maximum context length limits one-shot or few-shot learning learning [22]. Therefore, inference is only done under the zero-shot setting to prevent unnecessary truncation because the input length of the source code of a fuzzing function and its underlying functions can be very large.

4.3.1 Configuration

The inference parameters of the LLM can be configured by specifying a JSON file. The JSON configuration format is as follows:

```
{
  "do_sample": false,        # Ignored by OpenAI
  "temperature": 1.0,        # Default = 1.0, ignored if do_sample is false
  "top_p": 1.0,              # Default = 1.0, ignored if do_sample is false
  "diversity_penalty": 2.0,  # Ignored by OpenAI, requires group beam search
  "repetition_penalty": 2.0, # frequency_penalty for OpenAI
  "presence_penalty": 2.0,   # Ignored by HuggingFace
  "num_beams": 10,           # Ignored by OpenAI, default = 1 (no beam search)
  "num_beam_groups": 10      # Ignored by OpenAI, default = 1 (no group beam search)
}
```

We created 7 different configurations with commonly used temperatures [2] and different top_p values. In most configurations either the temperature or top_p is altered. The configuration files can be found in Appendix A. These have also been used during evaluation and are compared in the results. Additionally, the generation length and the number of return sequences can be specified in the command line parameters of the Python program.

The exact implementation of the data processor and parsing the output of the LLM differs for fine-tuned models and prompt-tuned base models, this is described in the following two sections. Nevertheless, the final output remains the same: initial seed files generated by the LLM that can be used by a fuzzer. To make it compatible with a wide variety of fuzzers, we used a common binary format for the seed files. This format is used in popular fuzzers such as AFL++ [63] and libFuzzer [40]. Moreover, the Go programming language supports an official tool to convert binary seed files to the format used by the built-in Go fuzzer [26].

4.3.2 Fine-tuned models

The implementation of the fine-tuned models consists of a straightforward approach. The data processor simply adds the split token used during training to the output of the source code parser (except for seq2seq models). Following the execution of the model, duplicate generated outputs are removed before storing the outputs directly in seed files. The fine-tuned GPT-3.5 Turbo model is processed as a prompt-tuned model due to the necessity of specific prompts.

4.3.3 Prompt-tuning base models

To experiment with the selected LLMs that could not be fine-tuned, such as GPT-4, and to compare the fine-tuned models with their base variants prior to fine-tuning, we also utilized

prompt-tuning, a method that bypasses the need for model retraining. Prompt-tuning has shown promising results, surpassing general fine-tuning in code intelligence tasks, especially when lacking task-specific data [69].

The prompt-tune parameters can be configured in a separate JSON file. The prompt-tuning configuration format is as follows:

```
{
  "prefix": "You are a code completer.\n",
  "suffix": "\n```\nfunc Test<count>Bugs() {\n\tinputs := []string{",
  "stop": "}",                    # Optional stop token (next to EOS token)
  "multi_vals": true,             # Extract multiple values per line
  "code_only": false              # Set to true for code-only models
}
```

The data processor adds the prefix and suffix to the output of the source code parser and replaces `<count>` with the desired seed count specified by a command-line parameter for prompts that generate multiple seeds in one return sequence. Additionally, a stop token can be specified to stop execution when this token is generated. The generated seeds are extracted from the outputs of the model by parsing each line and searching for string values. The data processor attempts to decode any escaped characters or bytes before storing the values in seed files.

For code-only models trained solely on source code, an option was added to the data processor and source code parser so that the model only receives source code as input. Enabling this option removes code block encapsulation and package names are added as comments.

We created various prompt-tune configurations that overall seem to perform well based on manual testing on various models (Appendix A). These configurations have also been used during evaluation and are compared in the results.

4.4 Evaluation

We evaluated our approach to demonstrate its effectiveness in improving the performance of state-of-the-art fuzzing using our implementation with various LLMs. Evaluating fuzzing performance is not a straightforward task. Commonly used metrics for evaluating machine learning models in fuzzing are accuracy, precision, recall, and loss [71]. However, these metrics do not measure actual fuzzing performance. Commonly used metrics for evaluating fuzzing performance are code coverage, unique code paths, unique crashes or bugs, and pass rate [6, 71]. But previous research indicates that these metrics are not sufficient to measure true fuzzing performance. There may be a strong correlation between the coverage achieved and the number of bugs found by a fuzzer [6], but another paper found a weak correlation between the number of crashes and ground-truth bugs [30]. Both papers conclude that higher coverage does not necessarily mean better fuzzing performance.

Magma [30], a ground-truth fuzzing benchmark in C, introduced three bug-centric metrics to better measure fuzzing performance. These metrics are the number of times a bug is reached, triggered, and detected in a certain amount of time. The first metric is increased when the part of the code containing the bug is reached. When the bug is also triggered, the second metric is increased. Finally, if the bug is not triggered but only detected by the fuzzer (e.g. memory corruption, data races, other unexpected bugs), then the last metric is increased.

While Magma is an excellent benchmark for C fuzzers, it only provides a dataset in C and does not support the built-in Go fuzzer. This limitation meant that we could not directly use Magma for our evaluation because our fine-tuned models are fine-tuned for Go. Furthermore, there is a risk that an LLM might have encountered the dataset provided by Magma during its pre-training phase, causing overly optimistic results.

To mitigate this risk and improve our evaluation, we manually crafted a new dataset for testing consisting of three diverse test cases varying in complexity. Similar to the creation of our validation dataset, we manually (re-)introduced unique bugs to existing Go code. To reduce the risk of the LLM encountering the same code it had been trained on, we only included existing code from Ultraware's private repositories and its dependencies. We did not include code depending on popular external libraries. This approach should result in a fairer comparison.

We also manually crafted realistic fuzzing tests for the functions containing the bugs. The fuzzing tests were designed such that bug-centric metrics could be collected without Magma. A crash means that the bug has been triggered. Additionally, we added instrumentation at the bug's location to capture when a bug is reached. However, this is a subjective metric since a bug can be fixed in multiple locations. Given the large number of bug examples required for these metrics, we also collected coverage metrics from the fuzzer, although no strong conclusions can be drawn from these [6, 30]. Finally, we collected "detected" bugs as unexpected bugs, which are crashes caused by an input that does not contain the elements of the expected input that should fail the fuzzing test.

The source code and a short description of the fuzzing test cases, including the function containing the bug, can be found in Appendix B. It also describes the expected inputs to cause a crash.

The evaluation of our approach was done with libFuzzer (16.0.6) [40]. We did not evaluate our approach with the built-in Go fuzzer since it is still in its early stages, therefore it does not perform as well as libFuzzer [77]. Furthermore, we did not consider AFL++ because it does not officially support Go. There are some workaround possible to make it work through gccgo, but this is highly experimental and not stable to use.

We began our evaluation by collecting baseline metrics. We ran the fuzzer 28 times per test case without any initial seed generation to match the minimum number of runs required to test each configuration combination for the prompt-tune models. Next, we ran the fuzzer the same number of times with a few manually crafted seeds, such as valid IBANs when fuzzing the iban test case. After collecting the baseline metrics, we generated initial seed files before each run with our implementation using the fine-tuned models. We ran the fuzzer 4 times for each of the 7 configurations in Appendix A. Finally, we ran the fuzzer for each of the 28 inference-prompt-tune configuration combinations with seeds generated by the prompt-tuned models.

For each run, 10 seeds were generated by setting the number of return sequences to 10 for the fine-tuned models and single-line prompt-tune configurations. The generation length was limited to 200 to speed up generation. For the multi-value prompt-tune configurations, the number of return sequences was set to 2 to preserve the effect of inference configuration parameters. This was not set higher to further utilize multi-value generation by setting the desired seed count parameter. This parameter was set to 5 to match the number of seeds generated by the fine-tuned models for a fairer comparison. Because multiple seeds can be generated in one return sequence, the seed generation length was limited to 2000.

During seed generation, some models were also executed with limited context length to prevent out-of-memory errors on a single A100 80 GB GPU despite utilizing memory-saving techniques. The used context lengths can be found in Table 14 in the appendix.

The variations in context lengths cause variations in seed generation times. Furthermore, some smaller models were executed on the fuzzing machine itself. The seed generation time mostly depends on the GPU, while the fuzzing process itself mostly depends on the CPU. Therefore, we did not include the seed generation time in the results. These are only shown per model for configuration comparison and in the appendix.

LibFuzzer was run as a single process with value profiling enabled. The test cases were compiled on Go 1.21.1 using golang-fuzz[*] with go-libfuzz-build[†] to automate coverage instrumentation for libFuzzer. For the iban test case, instrumentation was limited to the program itself because the code logic does not depend on any (external) libraries. This approach causes the fuzzer to skip irrelevant parts, which increases fuzzing performance. The specifications of the fuzzing machine are described in the table below.

Table 4: Hardware specifications fuzzing machine

Model	Dell Precision 7720
CPU	Intel i7-7920HQ @ 4.100GHz
Memory	16 GB DDR4 @ 2400 MHz
Disk	Samsung SSD 850 1TB

[*] `https://github.com/ultraware/golang-fuzz`
[†] `https://github.com/elwint/go-libfuzz-build`

5 Results

This section presents the fine-tuning results and our fuzzing performance evaluation of the fine-tuned models and prompt-tuned base models.

5.1 Fine-tuning

The fine-tuning results of the models are presented below in separate tables for each model. It is presented separately because the cross entropy loss may be computed differently depending on the model's architecture. Each row corresponds to the best epoch of the tested configuration based on the lowest validation loss. The table also presents the accuracy of predicting the next token in the training data. We prioritized validation loss over accuracy since multiple inputs can potentially trigger the same bug, thus the error distance presented by the loss offers a better indication of the model's performance. If no convergence was observed, only the last epoch is presented.

Table 5: Fine-tuning GPT-3.5/base results (max context length 4096)

Model	Epoch	Val Loss	Val Acc	Train Loss	Train Acc
GPT-3.5 Turbo PT *	2	1.090	0.611	1.297	0.737
GPT-3.5 Turbo	2	2.624	0.260	2.408	0.570
davinci-002	5	3.461	0.366	2.381	0.567

Table 5 presents the fine-tuning results of the closed-source GPT-3.5/base models. The learning rate is unknown and currently not configurable [52]. The PT version was fine-tuned with prompt-tuning as described in the methods section. The loss and accuracy may seem significantly better compared to the open-source models below. However, unlike our loss calculation for the open-source models, OpenAI may also include predicting the prompt tokens for its loss calculation during fine-tuning. The legacy API included a weight to use for loss on the prompt tokens [52]. However, this weight is unknown for newer models. This means that its loss is not directly comparable to that of other models.

For the open-source models below, although we tested many various training configurations, only the top two results per model type and size are presented.

Table 6: Fine-tuning StarCoder(Base) 15.5B results (bf16 with max context length 6528, LoRA enabled)

Model	Epoch	Learning Rate	Val Loss	Val Acc	Train Loss	Train Acc
StarCoder	5	3.0×10^{-5}	3.610	0.525	1.687	0.300
StarCoder	22	5.0×10^{-6}	3.752	0.475	2.519	0.227
StarCoderBase	2	5.0×10^{-5}	4.024	0.475	3.158	0.215
StarCoderBase	3	3.0×10^{-5}	4.045	0.475	2.954	0.218

StarCoder is a fine-tuned version of StarCoderBase for Python. Although our training data consists of Go code, StarCoder notably fine-tuned better than StarCoderBase on our training data.

Table 7 below presents the fine-tuning results of the various CodeGen-Multi model sizes. Interestingly, it seems that the largest CodeGen-Multi models fine-tuned worse on our data

*The actual loss may be slightly higher due to the included additional prompt tokens used for prompt-tuning.

compared to the smaller variants. This may be caused by the increased complexity of training larger models, which may require longer or other fine-tuning strategies.

Table 7: Fine-tuning CodeGen-Multi results (fp16 with max context length 2048)

Size	LoRA	Epoch	Learning Rate	Val Loss	Val Acc	Train Loss	Train Acc
2B	Yes	2	3.0×10^{-4}	4.461	0.167	2.419	0.243
16B	Yes	2	3.0×10^{-4}	4.464	0.188	2.505	0.258
350M	No	2	3.0×10^{-6}	4.476	0.229	2.319	0.257
16B	Yes	3	3.0×10^{-5}	4.542	0.125	3.741	0.152
2B	Yes	2	1.0×10^{-3}	4.557	0.188	2.781	0.270
350M	Yes	8	3.0×10^{-5}	4.672	0.167	3.316	0.154
6B	Yes	4	3.0×10^{-5}	4.818	0.208	3.654	0.167
6B	Yes	2	3.0×10^{-4}	4.876	0.104	2.209	0.263

The CodeT5+ 770M and 220M models have a default maximum context length of 512, but a context length of 2048 is also supported. Because there was no significant difference between a context length of 512 and a context length of 2048 during fine-tuning, only the results with a context length of 2048 are presented in the table below.

Table 8: Fine-tuning CodeT5+ results (fp16 with max context length 2×2048)

Size	LoRA	Epoch	Learning Rate	Val Loss	Val Acc	Train Loss	Train Acc
770M	No	3	1.05×10^{-5}	2.589	0.511	1.097	0.647
220M	No	8	3.0×10^{-6}	2.788	0.500	1.732	0.571
220M	No	1	3.0×10^{-5}	2.834	0.500	2.535	0.471
770M	No	12	1.0×10^{-6}	2.917	0.522	1.821	0.600
16B	Yes	2	3.0×10^{-4}	4.682	0.271	1.824	0.619
6B	Yes	2	3.0×10^{-4}	5.233	0.229	2.228	0.575
2B	Yes	2	3.0×10^{-4}	5.335	0.167	2.297	0.538
16B	Yes	5	3.0×10^{-5}	5.372	0.250	2.619	0.583
2B	Yes	7	3.0×10^{-5}	5.380	0.188	2.342	0.540
6B	Yes	5	3.0×10^{-5}	5.629	0.229	3.026	0.549

The 770M model unexpectedly fine-tuned significantly better than the 16B model. This difference may be caused by the architecture of the smaller models being fundamentally different than the larger models as described in the background section.

Finally, the results of fine-tuning the CodeGen2.5 models are presented in Table 9. Because StarCoder fine-tuned better than StarCoderBase, we were also interested in fine-tuning CodeGen2.5-mono, the fine-tuned version of CodeGen2.5 for Python.

Table 9: Fine-tuning CodeGen2.5 7B results (max context length 2048, LoRA enabled)

Model	Type	Epoch	Learning Rate	Val Loss	Val Acc	Train Loss	Train Acc
mono	bf16	2	3.0×10^{-4}	3.939	0.229	2.676	0.224
multi	bf16	2	3.0×10^{-4}	4.569	0.146	2.605	0.228
mono	bf16	4	3.0×10^{-5}	4.861	0.125	3.366	0.183
multi	bf16	4	3.0×10^{-5}	4.920	0.104	3.713	0.120
multi	fp16	4	3.0×10^{-5}	4.924	0.104	3.768	0.114
mono	fp16	4	3.0×10^{-5}	5.069	0.083	3.426	0.188

Similar to StarCoder, the mono version of CodeGen2.5 fine-tuned better than the multi

version. We also found that the bfloat16 floating-point format works best with CodeGen2.5.

However, the fine-tuning results do not present a good indication of its effectiveness in improving seed generation in fuzzing. The next section describes the evaluation results comparing the effectiveness of the fine-tuned models and the prompt-tuned base models.

5.2 Evaluation and prompt-tuning

This section presents an analysis of the collected metrics during fuzzing. The results are illustrated per method: fuzzing without seeds (*base*), fuzzing with manually crafted seeds (base-corp), fuzzing with seeds generated by fine-tuned models (ft), and fuzzing with seeds generated by prompt-tuned models (pt). In total libFuzzer has run 84 times per method (28 samples per test case) with a timeout of 10 minutes per run. For the fine-tuned models, three duplicate results per test case have been removed (because of turned-off sampling generation) resulting in 25 samples per test case.

To determine the improved (or decreased) fuzzing performance of a method compared to the baseline, several two-tailed hypothesis tests are performed with significance level $\alpha = 0.05$. The results are divided into two groups: short fuzzing performance of 30 seconds and longer fuzzing performance of 10 minutes, which is the timeout of the fuzzing runs. The results do not include seed generation times as mentioned in the methods section. These are only shown per model for configuration comparison and in the appendix.

Figure 4 presents the total percentage of runs when a bug was triggered (crash) during fuzzing. Figure 5 presents the percentage of runs when a bug was triggered per test case. The methods in bold perform significantly better ($p < 0.05$) compared to the baseline when fuzzing for a maximum of 30 seconds. This is based on a two-proportion z-test, given the proportion of bugs triggered with the best method p_m and the proportion of bugs triggered with the baseline p_b with $H_0 : p_m = p_b$ and $H_a : p_m \neq p_b$.

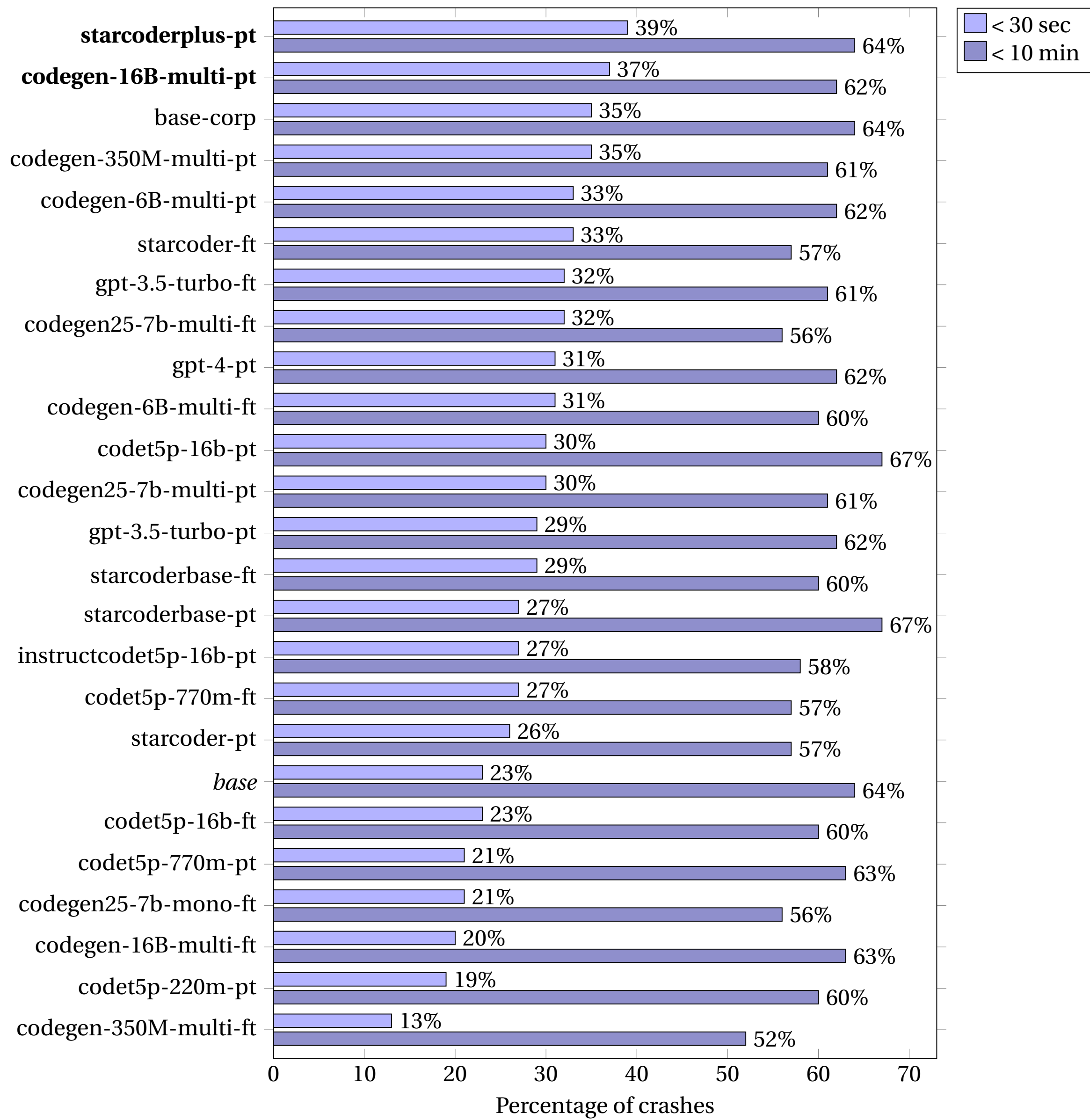


Figure 4: Total percentage of runs with a triggered bug (crash)

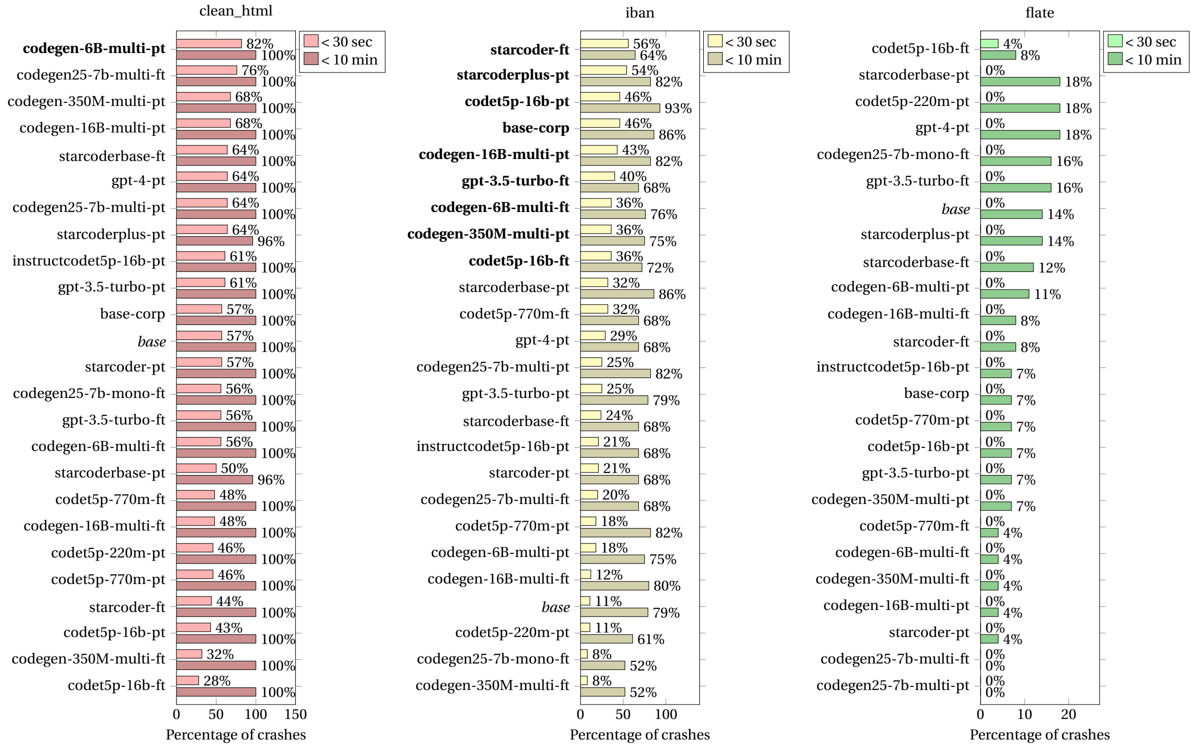


Figure 5: Percentage of runs with a triggered bug (crash) per test case

There is no significant difference ($p \geq 0.05$) between the baseline and other methods for the number of triggered bugs within 10 minutes of fuzzing. This is also the case when looking at the best results per test case. In fact, there is only a significant difference ($p < 0.05$) for the two worst iban and flate methods, meaning they performed worse than the baseline when fuzzing for a maximum of 10 minutes. Therefore, there is not enough evidence to suggest that the tested methods can discover significantly more bugs within 10 minutes of fuzzing compared to the baseline.

However, starcoderplus-pt and codegen-16B-multi-pt perform significantly better compared to the baseline when fuzzing for only a maximum of 30 seconds. This suggests that fuzzing with seeds generated by an LLM is more effective for shorter runs. However, this is excluding seed generation time, and the number of crashes is constrained by the number of runs. Therefore, it is also essential to consider the time it took to discover each bug and the type of bug, especially for the least and most complex test cases.

The data containing the time until a bug is triggered contains timed-out runs. These are right-censored observations, meaning that they are only partially known (the time is at least 10 minutes for the timed-out runs). A common method to include right-censored data for comparison is using survival analysis by performing a two-tailed logrank test [4] comparing the survival curves using the Kaplan-Meier estimator [33] to model the fuzzer's survival function. These curves represent the probability of not discovering a bug over time. The Kaplan-Meier estimator has also been used in the Magma fuzzing benchmark [30].

Given the survival curve of a method $S_m(t)$ and the survival curve of the baseline $S_b(t)$ at time t, performing a log-rank test with $H_0 : S_m(t) = S_b(t)$ and $H_a : S_m(t) \neq S_b(t)$, does not reject the null hypothesis ($p \geq 0.05$) when comparing all test cases combined (for every method) with a maximum fuzzing time of 10 minutes. However, there is a significant difference ($p < 0.05$) for

some methods when comparing the survival curves of the clean_html and flate test cases. The logrank test results of these methods together with the hazard ratio (HR) are presented in Table 10. The hazard ratio is calculated using the Cox proportional hazards model, a common method to quantify the distance between the curves [50].

Table 10: Logrank test with 10 minutes fuzzing time

Test Case	Method	Test Statistic	P-value	Hazard Ratio
clean_html	codegen-350M-multi-ft	6.471	0.011	0.483
clean_html	codegen25-7b-multi-ft	6.748	0.009	2.068
clean_html	codet5p-16b-ft	14.273	< 0.001	0.321
clean_html	starcoderbase-pt	5.27	0.022	0.516
iban	codet5p-16b-pt	10.084	0.001	2.500

The table reveals that only a few methods show a significant improvement in triggering a bug faster. In fact, the methods with an hazard ratio in red (smaller than 1) perform significantly worse compared to the baseline for the clean_html test case. Only codegen25-7b-multi-ft and codet5p-16b-pt perform significantly better compared to the baseline for the clean_html and iban test cases respectively. The hazard ratio is larger than 2, meaning that fuzzing using these methods compared to the baseline results in triggering a bug at least twice as likely over the entire fuzzing period. Figure 6 presents the survival curves of these methods together with the confidence intervals.

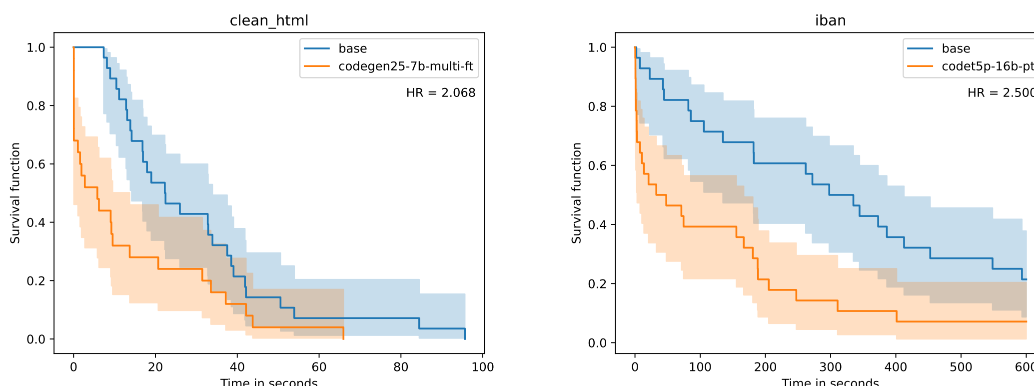


Figure 6: Kaplan-Meier survival curves of best-performing methods within 10 minutes

Most of the advantage is observed in the first few seconds, where the model generated seeds that trigger a bug (almost) instantly. This is important when taking into account seed generation time.

Unfortunately, a significant improvement in overall performance is only present when right-censoring the data to only 30 seconds. Table 11 reveals that more methods per test case perform significantly better when limiting the fuzzing time to 30 seconds.

Table 11: Logrank test with 30 seconds fuzzing time

Test Case	Method	Test Statistic	P-value	Hazard Ratio
All	codegen-16B-multi-pt	5.679	0.017	1.977
All	starcoderplus-pt	6.772	0.009	2.083
clean_html	codegen25-7b-multi-ft	7.53	0.006	2.508
clean_html	codegen-6B-multi-pt	4.607	0.032	1.993
clean_html	codet5p-16b-ft	4.601	0.032	0.390
iban	starcoder-ft	13.612	< 0.001	7.456
iban	codegen-16B-multi-pt	7.921	0.005	5.127
iban	starcoderplus-pt	12.569	< 0.001	6.926
iban	codegen-350M-multi-pt	4.914	0.027	3.876
iban	codegen-6B-multi-ft	4.927	0.026	3.945
iban	gpt-3.5-turbo-ft	6.543	0.011	4.634
iban	codet5p-16b-pt	9.212	0.002	5.628
iban	codet5p-16b-ft	5.088	0.024	4.018
iban	base-corp	9.29	0.002	5.663
iban	starcoderbase-pt	4.157	0.041	3.572

Figure 7 presents the survival curves of the best overall methods when fuzzing for only 30 seconds. Most of the advantage is again observed in the first few seconds. The overall advantage of these methods becomes less significant when fuzzing for a longer period with the hazard ratio approaching 1 (see Figure 19 in appendix).

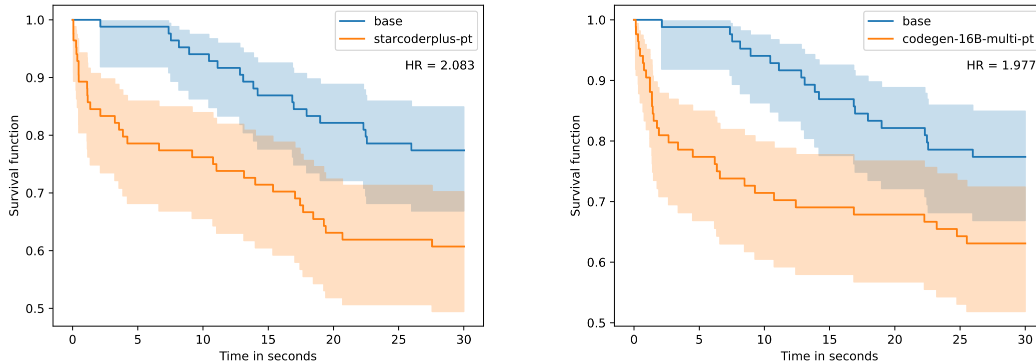


Figure 7: Kaplan-Meier survival curves of best-performing methods within 30 seconds

There is a relationship between the early triggering of bugs and the time it takes for a fuzzer to reach the location of the bug. Figure 8 presents the average time it took until the location of the iban bug was reached. Only the iban test case is shown because the differences in the other test cases are negligible (see Table 15 in the appendix).

Because all runs reached the bug location and the variances are unequal, a Welch's t-test was performed. Given the mean of a method μ_m and the mean of the baseline μ_b, performing a two-tailed Welch's t-test with $H_0 : \mu_m = \mu_b$ and $H_a : \mu_m \neq \mu_b$, rejects the null hypothesis for the methods in bold ($p < 0.05$).

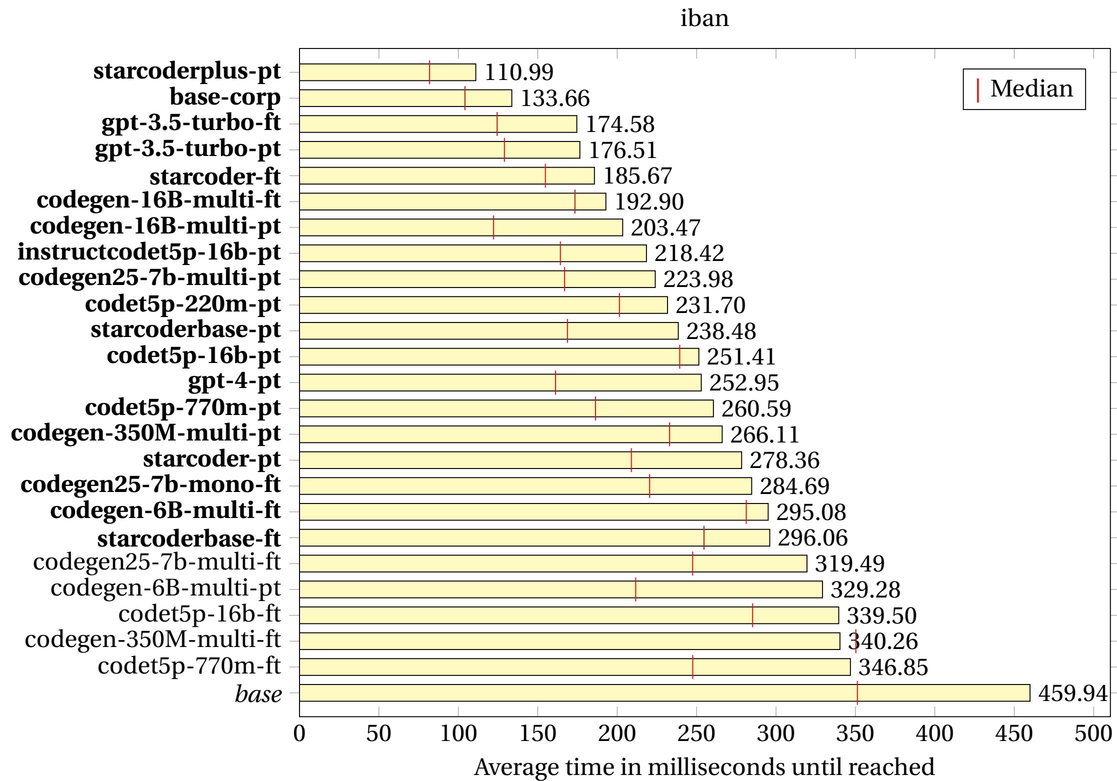


Figure 8: Average time until iban bug location reached

In most runs the bug location was reached within less than half a second, which suggests that the introduced bugs in the test cases are not very deep. This does not necessarily mean that the bugs are easy to find. For instance, the flate test case has lower reached times but the bug was not quickly triggered. Determining when a bug location is reached is subjective because the bug can be fixed at multiple points in the code.

Nevertheless, triggering a bug faster is not the only important aspect of fuzzing. Some runs triggered a bug with an unexpected input. These inputs did not contain the elements of the expected input that should fail the fuzzing test. The generated seeds of some models increased the likelihood of the fuzzer creating an unexpected input that triggers a bug, as seen in the following figures. This is interesting since LLMs are able to generate diverse inputs targeting different potential bugs, and not just generating inputs for one specific potential bug. This increases the fuzzer's ability to discover a wider variety of bugs.

When performing the same two-proportion z-test, the difference compared to the baseline is significant for the methods in bold in the following figures.

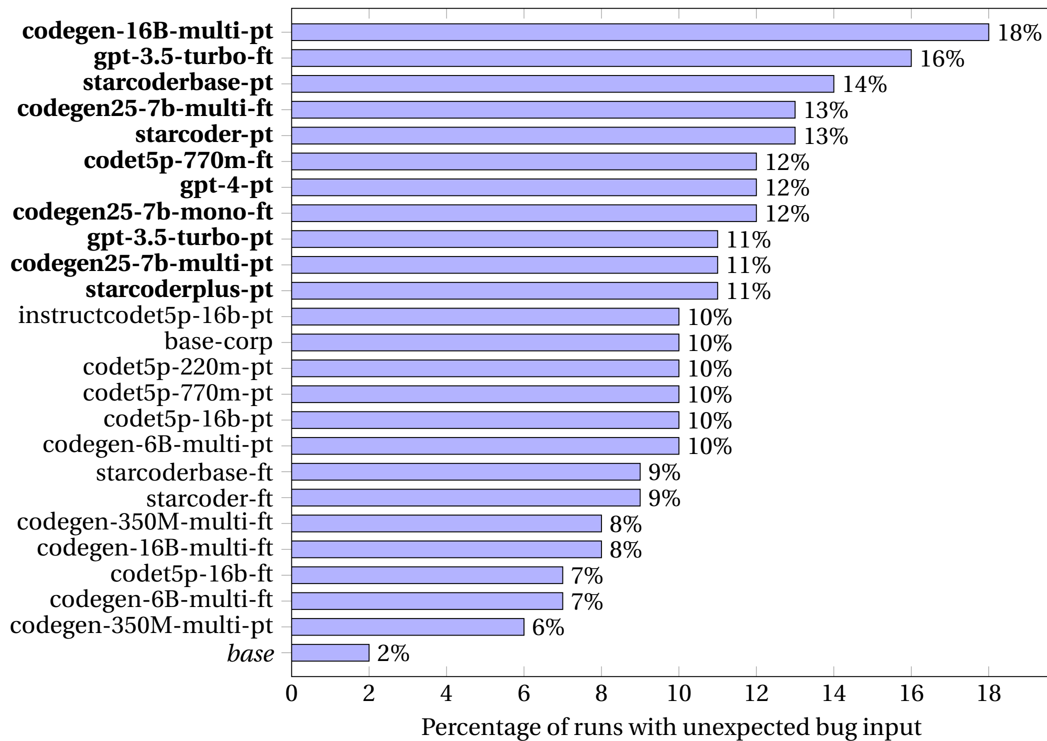


Figure 9: Total percentage of runs with an unexpected input triggering a bug

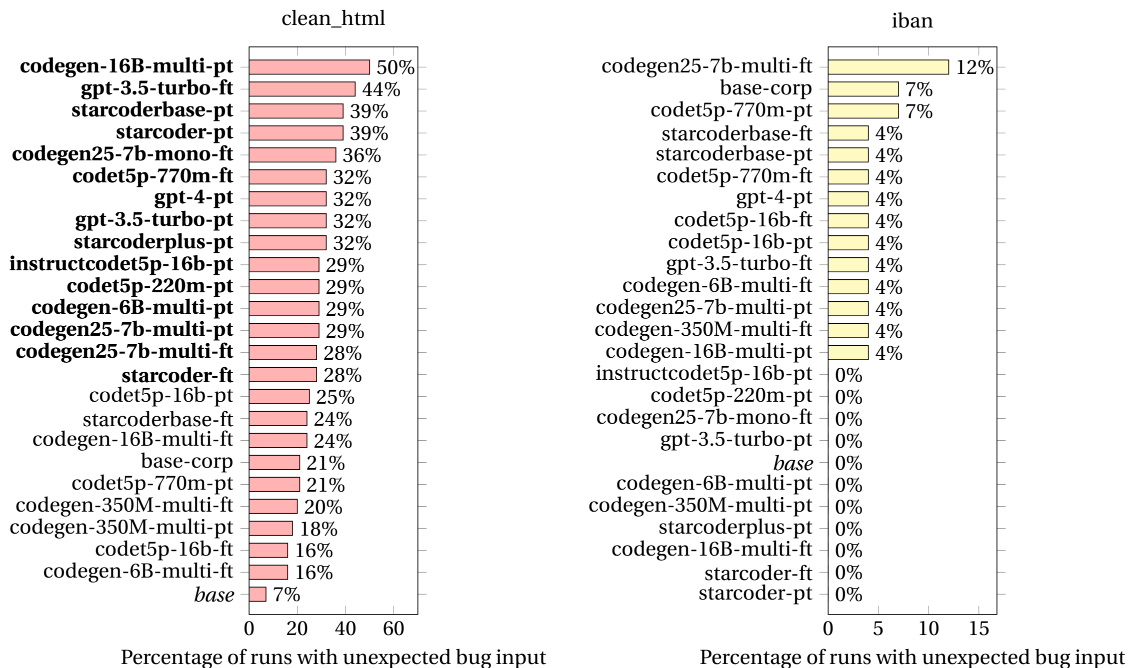


Figure 10: Percentage of runs with an unexpected input triggering a bug per test case

Interestingly, the results show that in most cases the fine-tuned models perform worse than the prompt-tuned models. The best overall open-source models are prompt-tuned, while their fine-tuned variants show worse performance. For the closed-source models, the fine-tuned GPT-3.5 Turbo model performs slightly better than the prompt-tuned version, although the difference is not significant. This may be caused by some limitations we encountered during fine-tuning. This is further discussed in the conclusions section.

5.3 Best model configurations and coverage

The quality of the seeds generated by the model depends on the used generation configuration. The best configuration differs for each model and can be different for each fuzzing test case. The results above indicate that the prompt-tuned CodeGen-Multi 16B model is the best overall model for seed generation for Go based on our results. The used configurations for this model are compared in the tables below. The content of the configuration files can be found in Appendix A. Finally, the reached fuzzing coverage is compared to the baselines (with and without manually crafted seeds) per test case. The configuration comparison and coverage of the other best-performing models, including smaller models, can be found in Appendix D.

Table 12: Model configuration results for codegen-16B-multi-pt

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds
temp_0.6.json	8	4	86 ms	34.37 s
top_p_0.99.json	8	3	44 ms	29.3 s
temp_0.8.json	8	1	34 ms	33.55 s
top_p_0.75.json	8	1	54 ms	35.53 s
diverse_beam_search.json	8	1	152 ms	70.89 s
temp_0.2.json	6	3	94 ms	29.0 s
top_p_0.50.json	6	2	108 ms	41.4 s

Table 13: Prompt-tune configuration results for codegen-16B-multi-pt

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds
text_multi.json	14	3	56 ms	47.99 s
text.json	14	3	67 ms	30.23 s
code.json	13	6	116 ms	31.73 s
code_multi.json	11	3	87 ms	46.64 s

In Figure 11 the reached coverage is compared by plotting the coverage of every run over time. For the iban test case, the number of features (which libFuzzer uses to evaluate coverage) is plotted instead because of low coverage. This was caused by the fact that no (external) libraries were instrumented for fuzzing the iban test case.

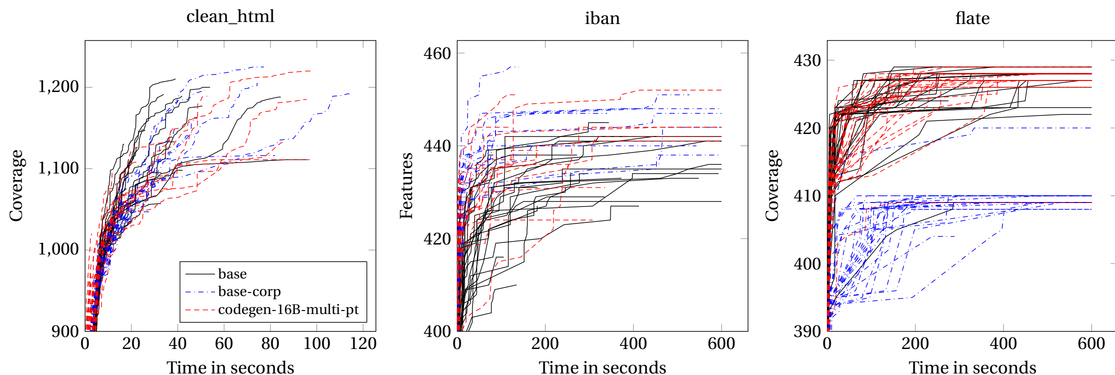


Figure 11: Reached coverage per test case

Although higher coverage does not necessarily mean better fuzzing performance [6, 30], the limited coverage of the baseline with manually crafted seeds is noticeably lower for the entire 10-minute fuzzing runs, while seeds generated by the model are not causing lower coverage.

This could be attributed to the test case being complex, making it difficult for the fuzzer to reach higher coverage when starting with seeds containing valid compression bytes. This means that perfectly well-formed seeds may actually reduce the effectiveness of fuzzing complex test cases, which has also been observed in previous research [17].

The coverage data is limited by the time taken to trigger a bug. Since most bugs are triggered early with the model-generated seeds, a more detailed view of the first few seconds is necessary to observe clearer differences. In Figure 12, the reached coverage is limited to the first three seconds of the runs.

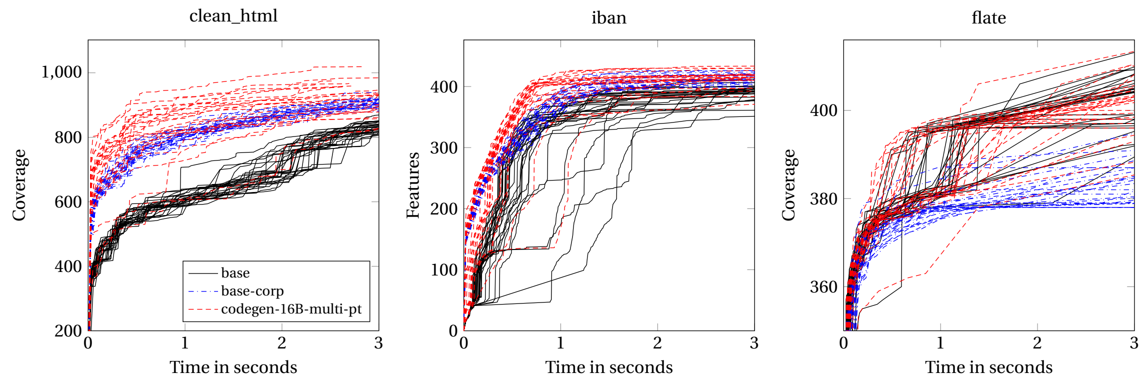


Figure 12: First three seconds of reached coverage

In the fuzzing runs where the model's generated seeds are used, a noticeable advantage in coverage is seen compared to the baselines in the clean_html and iban test cases. Most coverage is already reached in the first second. This corresponds with the survival curve of codegen-16B-multi-pt shown in Figure 7, where most of the advantage is observed in the first few seconds.

6 Conclusions and Future Work

The analyzed results show that the performance of state-of-the-art fuzzing in Go can be improved using Transformer-based large language models for initial seed generation. We found that overall, within the first 30 seconds of fuzzing, libFuzzer triggers a bug twice as likely with seeds generated by the best-performing LLMs compared to fuzzing without seeds. The shorter the fuzzing duration, the more significant this improvement becomes. Furthermore, we found significantly more unexpected bugs, which are crashes caused by an input that does not contain the elements of the expected input that should fail the fuzzing test. This revealed several new bugs in the libraries containing the code used for the test cases, meaning that libFuzzer can discover a wider variety of bugs more quickly when fuzzing with seeds generated by LLMs.

However, the significance of improvement heavily depends on the model used for seed generation, the fuzzing test, and the specifications of the fuzzing machine. Overall, we found that larger LLMs perform better, but since the advantage becomes less significant when fuzzing for a longer period, there is a larger advantage when fuzzing for a shorter period using a fast GPU to reduce seed generation times.

In practice, the fuzzing process can already start while generating seed files. The fuzzer mostly utilizes the CPU, while seed generation utilizes the GPU. When seed generation is completed, the fuzzing process can be restarted to use the generated seed files, or another fuzzing process can be started to retain the progress of the running fuzzer reaching potentially other parts of the program. Even when seed generation is slow, there could still be an advantage since we found a significant improvement in discovering unexpected bugs. This means that the fuzzer could discover bugs that may not be triggered when fuzzing without seed files.

Based on our results, the best-performing LLMs for seed generation in Go are the prompt-tuned CodeGen-Multi 16B and prompt-tuned StarCoderPlus models. Interestingly, these both outperform the closed-source GPT models, of which the fine-tuned GPT-3.5 Turbo PT version performs the best. Some smaller open-source models, such as the fine-tuned CodeGen2.5 7B multi and prompt-tuned CodeGen-Multi 350M models, do not show a significant improvement overall, but they can still have a positive impact on fuzzing performance depending on the fuzzing test.

Unfortunately, we failed to show any significant impact on fuzzing performance on fuzzing tests requiring inputs that are uncommonly found in source code or the training dataset of LLMs, such as compression bytes in our flate test case. We believe that our approach is more effective in improving fuzzing performance when the source code and fuzzing inputs are similar or related to the data the LLM was trained on. However, the insignificant improvement on our flate test case could also be a result of our small sample size, limited fuzzing time, or the limited number of generated seeds. Further research is required to confirm this by experimenting with more and longer fuzzer runs and seeds.

We also failed to show that fine-tuning the open-source models for seed generation using our training data further improves fuzzing performance. This may be caused by some limitations we encountered during fine-tuning. One of these limitations is the small dataset size that was used for fine-tuning the models. Another limitation is the use of common classification metrics based on the expected output during fine-tuning. These metrics are accuracy and loss, which did not correlate to the fuzzing performance of the models measured by triggered bugs and

code coverage. These metrics were not applied during fine-tuning because of the complexity of modifying the fine-tuning process of large language models and the resources required to do this.

The differences between the metrics used for fine-tuning and the measured fuzzing performance may be explained by arguing that seed generation in fuzzing is not a classification task, but rather a conditional generation task. Seed generation in fuzzing requires a degree of randomness, making it unsuitable for classification since multiple unique inputs can trigger the same code path. This means that fuzzing performance metrics are more relevant for evaluating and comparing models for DL-based seed generation in fuzzing. This may also explain why classification metrics are rarely presented in previous related studies [71].

However, the worse performance of the fine-tuned models could also be attributed to the fine-tuning method itself. The open-source models were fine-tuned differently than GPT-3.5 Turbo, where the PT version included embedded instructions within the prompt of each training example that works best for the model prior to fine-tuning. This was done after fine-tuning the open-source models because the fine-tuning API for GPT-3.5 Turbo was released later. Due to time constraints, the open-source models have not been fine-tuned with embedded instructions in the training data. Further research is required to see if this also positively affects the open-source models to generate higher-quality seeds to further improve fuzzing performance.

Apart from fine-tuning, this study is also limited to the programming language Go. Our implementation should also work with other programming languages, but we do not know to what extent our implementation improves fuzzing performance in other programming languages. We expect similar results in other programming languages if the model supports the used programming language. However, there are significantly more fuzzers and LLMs that work with other programming languages. Therefore, further research is required to know the effectiveness of our implementation with other programming languages. It would especially be interesting to see if our implementation could have a larger impact on fuzzing performance in more popular languages where fuzzing is more common, such as C. There are also larger datasets available in other programming languages such as C or Java that can be used for fine-tuning the models for seed generation. Using these datasets during fine-tuning may result in a different outcome compared to our dataset in Go.

In conclusion, our approach improves the performance of state-of-the-art fuzzing in Go most effectively when seed generation is done next to fuzzing without seeds. Since LLMs are currently an active research topic and newer models are frequently released, the effectiveness of our approach may increase in the future when improved models are used in our implementation. Newer models can easily be configured in our implementation, although fuzzing performance needs to be re-evaluated when using these models.

References

[1] Joshua Ackerman and George Cybenko. "Large Language Models for Fuzzing Parsers (Registered Report)". In: *Proceedings of the 2nd International Fuzzing Workshop*. 2023, pp. 31–38.

[2] Ben Athiwaratkun et al. "Multi-lingual Evaluation of Code Generation Models". In: *arXiv preprint arXiv:2210.14868* (2022).

[3] Sid Black et al. "GPT-NeoX-20B: An Open-Source Autoregressive Language Model". In: *arXiv preprint arXiv:2204.06745* (2022).

[4] J Martin Bland and Douglas G Altman. "The logrank test". In: *Bmj* 328.7447 (2004), p. 1073.

[5] Marcel Böhme, Valentin JM Manès and Sang Kil Cha. "Boosting fuzzer efficiency: An information theoretic perspective". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 678–689.

[6] Marcel Böhme, László Szekeres and Jonathan Metzman. "On the Reliability of Coverage-Based Fuzzer Benchmarking". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1621–1633.

[7] Rishi Bommasani et al. "On the opportunities and risks of foundation models". In: *arXiv preprint arXiv:2108.07258* (2021).

[8] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[9] Nghi DQ Bui et al. "CodeTF: One-stop Transformer Library for State-of-the-art Code LLM". In: *arXiv preprint arXiv:2306.00029* (2023).

[10] Federico Cassano et al. "MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation". In: *IEEE Transactions on Software Engineering* (2023).

[11] Saikat Chakraborty et al. "Deep learning based vulnerability detection: Are we there yet". In: *IEEE Transactions on Software Engineering* (2021).

[12] Bei Chen et al. "CodeT: Code Generation with Generated Tests". In: *arXiv preprint arXiv:2207.10397* (2022).

[13] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *arXiv preprint arXiv:2107.03374* (2021).

[14] Tianqi Chen et al. "Training Deep Nets with Sublinear Memory Cost". In: *arXiv preprint arXiv:1604.06174* (2016).

[15] Liang Cheng et al. "Optimizing Seed Inputs in Fuzzing with Machine Learning". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 244–245.

[16] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[17] Yinlin Deng et al. "Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries". In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society. 2023, pp. 830–842.

[18] Yinlin Deng et al. “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*. 2023, pp. 423–435.

[19] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[20] Zhangyin Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[21] Daniel Fried et al. *InCoder: A Generative Model for Code Infilling and Synthesis*. 2023. arXiv: `2204.05999 [cs.SE]`.

[22] Tianyu Gao, Adam Fisch and Danqi Chen. “Making Pre-trained Language Models Better Few-shot Learners”. In: *arXiv preprint arXiv:2012.15723* (2020).

[23] Jonas Gehring et al. “Convolutional Sequence to Sequence Learning”. In: *CoRR* abs/1705.03122 (2017). arXiv: `1705.03122`. URL: `http://arxiv.org/abs/1705.03122`.

[24] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pp. 70–76.

[25] Google. *Fuzzer benchmarking as a service*. URL: `https://google.github.io/fuzzbench/` (visited on 17/08/2023).

[26] Google. *Go fuzzing*. URL: `https://go.dev/security/fuzz/` (visited on 01/08/2023).

[27] Google. *Go Vulnerability Database*. URL: `https://go.dev/security/vuln/database` (visited on 17/04/2023).

[28] Google. *TPU Research Cloud*. URL: `https://sites.research.google/trc/about/` (visited on 08/05/2023).

[29] Hazim Hanif et al. “The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches”. In: *Journal of Network and Computer Applications* 179 (2021), p. 103009.

[30] Ahmad Hazimeh, Adrian Herrera and Mathias Payer. “Magma: A ground-truth fuzzing benchmark”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (2020), pp. 1–29.

[31] Edward J Hu et al. “LoRA: Low-Rank Adaptation of Large Language Models”. In: *arXiv preprint arXiv:2106.09685* (2021).

[32] HuggingFace. *Transfosrmers*. URL: `https://huggingface.co/docs/transformers/main/en/index` (visited on 10/05/2023).

[33] Edward L Kaplan and Paul Meier. “Nonparametric Estimation from Incomplete Observations”. In: *Journal of the American statistical association* 53.282 (1958), pp. 457–481.

[34] Michael O Lam et al. “Automatically Adapting Programs for Mixed-Precision Floating-Point Computation”. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 2013, pp. 369–378.

[35] Jun Li, Bodong Zhao and Chao Zhang. “Fuzzing: a survey”. In: *Cybersecurity* 1.1 (2018), pp. 1–13.

[36] Raymond Li et al. "StarCoder: may the source be with you!" In: (2023). arXiv: `2305.06161 [cs.CL]`.

[37] Zhen Li et al. "Vuldeepecker: A deep learning-based system for vulnerability detection". In: *arXiv preprint arXiv:1801.01681* (2018).

[38] Hongliang Liang et al. "Fuzzing: State of the art". In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218.

[39] Wei Lin and Saihua Cai. "An Empirical Study on Vulnerability Detection for Source Code Software based on Deep Learning". In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2021, pp. 1159–1160.

[40] LLVM. *LibFuzzer – A library for coverage-guided Fuzz Testing.* URL: `https://llvm.org/docs/LibFuzzer.html`.

[41] Shuai Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". In: *arXiv preprint arXiv:2102.04664* (2021).

[42] Ziyang Luo et al. "WizardCoder: Empowering Code Large Language Models with Evol-Instruct". In: *arXiv preprint arXiv:2306.08568* (2023).

[43] Chenyang Lyu et al. "Smartseed: Smart seed generation for efficient fuzzing". In: *arXiv preprint arXiv:1807.02606* (2018).

[44] Sourab Mangrulkar et al. *PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods.* `https://github.com/huggingface/peft`. 2022.

[45] Paulius Micikevicius et al. "Mixed Precision Training". In: *arXiv preprint arXiv:1710.03740* (2017).

[46] Anh Nguyen, Nikos Karampatziakis and Weizhu Chen. "Meet in the Middle: A New Pre-training Paradigm". In: *arXiv preprint arXiv:2303.07295* (2023).

[47] Nicole Nichols et al. *Faster Fuzzing: Reinitialization with Deep Neural Models.* 2017.

[48] Erik Nijkamp et al. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.* 2023. arXiv: `2203.13474 [cs.LG]`.

[49] Erik Nijkamp et al. "CodeGen2: Lessons for Training LLMs on Programming and Natural Languages". In: *arXiv preprint arXiv:2305.02309* (2023).

[50] Mikhail Nikulin, Hong-Dar Isaac Wu et al. *The Cox Model and Its Applications.* Springer, 2016.

[51] OpenAI. *Code completion.* URL: `https://platform.openai.com/docs/guides/code` (visited on 14/04/2023).

[52] OpenAI. *Fine-tuning.* URL: `https://platform.openai.com/docs/guides/fine-tuning` (visited on 20/10/2023).

[53] OpenAI. *GPT-4 Technical Report.* 2023. arXiv: `2303.08774 [cs.CL]`.

[54] OpenAI. *OpenAI Models.* URL: `https://platform.openai.com/docs/models` (visited on 02/07/2023).

[55] Tomás Osório. "Differences between Autoregressive, Autoencoding and Sequence-to-Sequence Models in Machine Learning". In: *Machine learning articles* (2020).

[56] Colin Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *The Journal of Machine Learning Research* 21.1 (2020), pp. 5485–5551.

[57] Sebastian Raschka. *Machine Learning and AI Beyond the Basics.* No Starch Press, 2023.

[58] Replit. *replit-code-v1-3b*. `https://huggingface.co/replit/replit-code-v1-3b`. May 2023.

[59] Teven Le Scao et al. “BLOOM: A 176B-Parameter Open-Access Multilingual Language Model”. In: *arXiv preprint arXiv:2211.05100* (2023).

[60] Kostya Serebryany. “OSS-Fuzz - Google’s continuous fuzzing service for open source software”. In: Vancouver, BC: USENIX Association, 2017.

[61] Noah Shinn, Beck Labash and Ashwin Gopinath. “Reflexion: Language Agents with Verbal Reinforcement Learning”. In: *arXiv preprint arXiv:2303.11366* (2023).

[62] Ari Takanen et al. *Fuzzing for software security testing and quality assurance.* Artech House, 2018.

[63] *The AFL++ Fuzzing Framework.* URL: `https://aflplus.plus/`.

[64] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).

[65] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[66] Yaëlle Vinçont, Sébastien Bardin and Michaël Marcozzi. “A Tight Integration of Symbolic Execution and Fuzzing (Short Paper)”. In: *International Symposium on Foundations and Practice of Security.* Springer. 2021, pp. 303–310.

[67] Dmitry Vyukov. *go-fuzz: randomized testing for Go.* URL: `https://github.com/dvyukov/go-fuzz`.

[68] Ben Wang and Aran Komatsuzaki. *GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model.* `https://github.com/kingoflolz/mesh-transformer-jax`. May 2021.

[69] Chaozheng Wang et al. “No More Fine-Tuning? An Experimental Evaluation of Prompt Tuning in Code Intelligence”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2022, pp. 382–394.

[70] Shibo Wang and Pankaj Kanwar. “BFloat16: The secret to high performance on Cloud TPUs”. In: *AI & Machine Learning - Google Cloud* (2019).

[71] Yan Wang et al. “A systematic review of fuzzing based on machine learning techniques”. In: *PloS one* 15.8 (2020), e0237749.

[72] Yue Wang et al. “CodeT5+: Open Code Large Language Models for Code Understanding and Generation”. In: *arXiv preprint arXiv:2305.07922* (2023).

[73] Yunchao Wang et al. “NeuFuzz: Efficient Fuzzing With Deep Neural Network”. In: *IEEE Access* 7 (2019), pp. 36340–36352.

[74] Hongwei Wei et al. “A context-aware neural embedding for function-level vulnerability detection”. In: *Algorithms* 14.11 (2021), p. 335.

[75] Chunqiu Steven Xia et al. “Universal Fuzzing via Large Language Models”. In: *arXiv preprint arXiv:2308.04748* (2023).

[76] Frank F Xu et al. “A Systematic Evaluation of Large Language Models of Code”. In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 2022, pp. 1–10.

[77] Khaled Yakdan. *Improvements In Golang Fuzzing (Golang 1.19)*. URL: `https://www.code-intelligence.com/blog/golang-fuzzing-1.19` (visited on 17/10/2023).

[78] Gen Zhang et al. “PTfuzz: Guided Fuzzing With Processor Trace Feedback”. In: *IEEE Access* 6 (2018), pp. 37302–37313.

[79] Qinkai Zheng et al. “CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X”. In: *arXiv preprint arXiv:2303.17568* (2023).

A Configurations

Inference configs

diverse_beam_search.json

```
"do_sample": false,
"temperature": 1.0,
"top_p": 1.0,
"diversity_penalty": 2.0,
"repetition_penalty": 2.0,
"presence_penalty": 2.0,
"num_beams": 10,
"num_beam_groups": 10
```

temp_t.json; $t \in \{0.2, 0.6, 0.8\}$

```
"do_sample": true,
"temperature": t,
"top_p": 1.0,
"diversity_penalty": 0.0,
"repetition_penalty": 1.0,
"presence_penalty": 1.0,
"num_beams": 1,
"num_beam_groups": 1
```

top_p_p.json; $p \in \{0.50, 0.75, 0.99\}$

```
"do_sample": true,
"temperature": 1.0,
"top_p": p,
"diversity_penalty": 0.0,
"repetition_penalty": 1.0,
"presence_penalty": 1.0,
"num_beams": 1,
"num_beam_groups": 1
```

Prompt-tune configs

code.json

````
"prefix": "You are a code completer.\n",
"suffix":
"\n```\nfunc TestBug() {\n\tinput := \"",
"stop": "\n",
"multi_vals": false,
"code_only": false
````

code_multi.json

````
"prefix": "Complete the code.\n",
"suffix":
"\n```\nfunc Test<count>Bugs() {\n\tinputs := []string{\"",
"stop": "}",
"multi_vals": true,
"code_only": false
````

text.json

```
"prefix": "Generate a bug input.\n",
"suffix": "\nA bug input: \"",
"stop": "\n",
"multi_vals": false,
"code_only": false
```

text_multi.json

```
"prefix": "Generate <count> bug inputs.\n",
"suffix": "\n<count> bug inputs:\n1. \"",
"multi_vals": true,
"code_only": false
```

code.json (code only)

```
"prefix": "",
"suffix":
"\n\nfunc TestBug() {\n\tinput := \"",
"stop": "\n",
"multi_vals": false,
"code_only": true
```

code_multi.json (code only)

```
"prefix": "",
"suffix":
"\n\nfunc Test<count>Bugs() {\n\tinputs := []string{\"",
"stop": "}",
"multi_vals": true,
"code_only": true
```

text.json (code only)

```
"prefix": "",
"suffix": "\n\n// A bug input: \"",
"stop": "\n",
"multi_vals": false,
"code_only": true
```

text_multi.json (code only)

```
"prefix": "",
"suffix": "\n\n// A list of <count> bug inputs:\n// 1. \"",
"stop": "func",
"multi_vals": true,
"code_only": true
```

gpt-3.5-turbo.json

```
"prefix": "Generate a bug input.",
"suffix": "",
"multi_vals": false,
"code_only": false
```

Table 14: Model-specific configurations

Model	FT Context Length	PT Context Length	PT Code Only
GPT-3.5 Turbo	4096	4096	No
GPT-4	N/A	8192	No
StarCoder	8192	8192	Yes
StarCoderBase	8192	8192	Yes
StarCoderPlus	N/A	8192	No
CodeGen-Multi 350M	2048	2048	Yes
CodeGen-Multi 6B	2048	2048	Yes
CodeGen-Multi 16B	1200	1200	Yes
CodeT5+ 220M	N/A	4096	Yes
CodeT5+ 770M	4096	4096	Yes
CodeT5+ 16B	4096	1050	Yes
InstructCodeT5+	N/A	1050	No
CodeGen2.5-Multi	2048	2048	Yes
CodeGen2.5-Mono	2048	N/A	N/A

B Source code evaluation test cases

The clean_html function sanitizes unsafe HTML elements, where we expect inputs containing '<<', '>>', '<', or '>' to crash the fuzzing test if it results in a script tag. The iban test case validates an IBAN, where we expect IBANs containing a hyphen character to cause a crash. Finally, the flate fuzzing test checks for a decompression bomb, where a too-high compression ratio causes a crash. We found with normal fuzzing that a small input with a minimum length of 19 bytes can crash this fuzzing test.

Only the relevant pieces of the source code are shown. The additional instrumentation at the bug's location is marked as a comment.

Listing 1: clean_html

```go
func Fuzz(v string) int {
    in := []types.Detail{
        {Information: v},
    }
    actual := item.CleanHTML(in)
    if strings.Contains(actual[0].Information, `<script>`) {
        panic(`unexpected script tag`)
    }
    if strings.ContainsRune(actual[0].Information, '<') || strings.ContainsRune(
        actual[0].Information, '>') {
        return 1
    }

    return 0
}

// Package: item
// Uses external bluemonday package
func CleanHTML(details []types.Detail) []types.Detail {
    p := bluemonday.Policy{}
    re := regexp.MustCompile(`(?i)\<br ?\/?\>`)

    cleaned := make([]types.Detail, len(details))
    for i, d := range details {
        d.Information = re.ReplaceAllString(d.Information, "\n")
        d.Information = p.Sanitize(d.Information)
        d.Information = html.UnescapeString(d.Information)
        cleaned[i] = d
        // Reached bug location
    }
    return cleaned
}
```

Listing 2: iban

```go
func Fuzz(v string) int {
    defer func() {
        v := recover()
        if v == `found special char in iban` {
            panic(v)
        }
    }()

```

```
9       if len(v) > 34 {
10          return -1
11      }
12
13      valid, _, _ := iban.IsCorrectIban(v)
14      if !valid {
15          return 0
16      }
17
18      v = runes.Remove(runes.In(unicode.Space)).String(v)
19      if strings.IndexFunc(v, special) != -1 {
20          panic("found special char in iban")
21      }
22
23      return 1
24  }
25
26  func special(r rune) bool {
27      return (r < 'A' || r > 'Z') && (r < 'a' || r > 'z') && (r < '0' || r > '9')
28  }
29
30  // Package: iban
31  // No external packages
32
33  // IsCorrectIban checks if the given iban number corresponds to the rules of a
        valid iban number and also
34  // returns a properly formatted iban number string.
35  func IsCorrectIban(iban string) (isValid bool, wellFormated string, err error) {
36      var ibanConfig ibanCountry
37      ibanValid := false
38      wellFormated = ""
39      if len(iban) >= 15 { // Minimum length for IBAN
40          // Clean up string
41          iban = strings.ToUpper(strings.ReplaceAll(iban, " ", ""))
42
43          // Split string up
44          passedChars := len(iban)
45          passedCode, passedChecksum, passedBban := splitIbanUp(iban)
46          ibanConfig = countryList[passedCode]
47
48          if ibanConfig.chars > 0 {
49              if ibanConfig.chars == passedChars {
50                  convertedIban := rearrangeIBAN(passedCode, passedChecksum,
                        passedBban)
51                  convertedIban = convertCharToNumber(convertedIban)
52
53                  if calculateModulo(convertedIban) == 1 {
54                      ibanValid = true
55                      wellFormated = splitTo4(iban)
56                  }
57              } else {
58                  return false, wellFormated, fmt.Errorf(
59                  "%w: length (%s) does not match configuration length (%s)",
60                  ErrInvalidIBAN,
61                  strconv.Itoa(passedChars),
62                  strconv.Itoa(ibanConfig.chars),
63                  )
```

```
64              }
65          } else {
66              return false, wellFormated, fmt.Errorf("%w: country <%s> is not in the
                    list", ErrInvalidIBAN, passedCode)
67          }
68      } else {
69          return false, wellFormated, fmt.Errorf("%w: passed <%s>", ErrInvalidIBAN,
                iban)
70      }
71      return ibanValid, wellFormated, nil
72  }
73
74  func calculateModulo(iban string) int {
75      exit := false
76      tempIBAN := ""
77      rest := 0
78      for !exit {
79          if len(iban) > 9 {
80              tempIBAN = iban[0:9]
81
82              value, err := strconv.Atoi(tempIBAN)
83              if err != nil || value < -1 {
84                  return -1
85              }
86              if tempIBAN[0] == '+' {
87                  return -1
88              }
89              // Reached bug location if value < 0
90
91              rest = (value % 97)
92              iban = strconv.Itoa(rest) + iban[9:]
93          } else {
94              tempIBAN = iban
95
96              value, err := strconv.Atoi(tempIBAN)
97              if err != nil {
98                  return -1
99              }
100
101             rest = (value % 97)
102             exit = true
103         }
104     }
105     return rest
106 }
```

Listing 3: flate

```
1   func Fuzz(compressed []byte) (ret int) {
2       defer func() {
3           v := recover()
4           if v == `too high ratio` {
5               panic(v)
6           }
7           if v != nil {
8               ret = -1
9           }
```

```
10      }()
11
12      decompressed := util.Decompress(compressed)
13      if len(decompressed)/len(compressed) > 100 {
14          panic(`too high ratio`)
15      }
16
17      return 0
18  }
19
20  // Package: util
21  // Uses flate package in standard library
22
23  // Decompress decompress []byte using flate
24  func Decompress(in []byte) []byte {
25      decompressor := flate.NewReader(bytes.NewReader(in))
26
27      buf := new(bytes.Buffer)
28      if _, err := io.Copy(buf, decompressor); err != nil {
29          panic(err)
30      }
31      _ = decompressor.Close()
32
33      // Reached bug location
34
35      return buf.Bytes()
36  }
```

C Seed generation times

The seed generation times presented below include the time taken for model initialization. For the GPT models, the generation times are measured by the duration of the API calls.

C.1 OpenAI's API

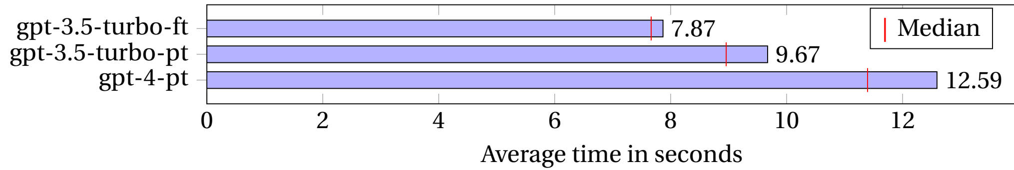


Figure 13: Average time to generate 10 seeds

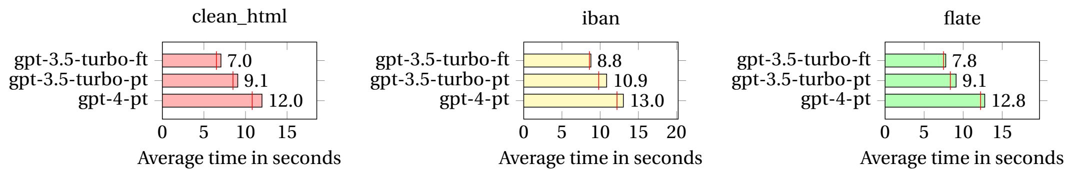


Figure 14: Average time to generate 10 seeds per test case

C.2 A100 GPU

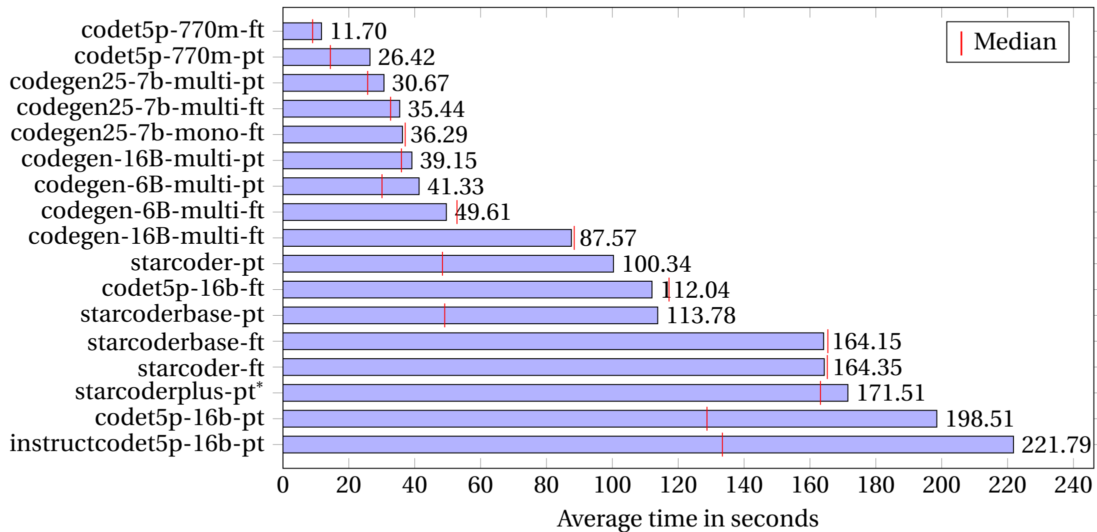


Figure 15: Average time to generate 10 seeds

*The generation times of starcoderplus-pt may have been affected by a bug causing the model to sometimes continue running after generating the seeds by missing the stop token. The seeds were not regenerated since it did not affect the output or fuzzing results because the output was truncated until the stop token. This bug was only present when this model was executed.

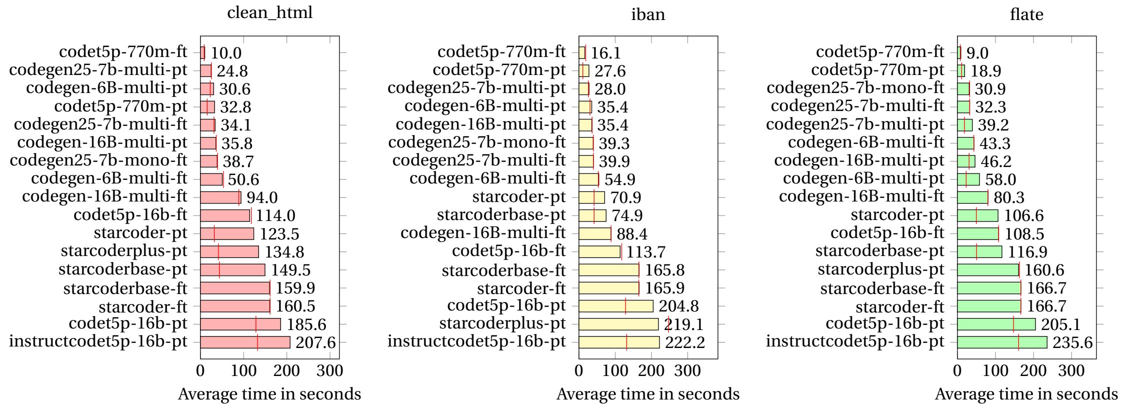


Figure 16: Average time to generate 10 seeds per test case

C.3 Fuzzing machine without GPU

The specifications of the fuzzing machine are described in Table 4.

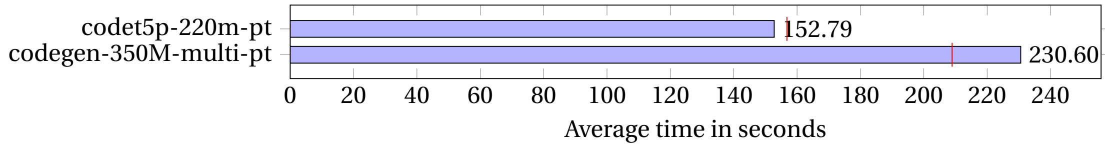


Figure 17: Average time to generate 10 seeds

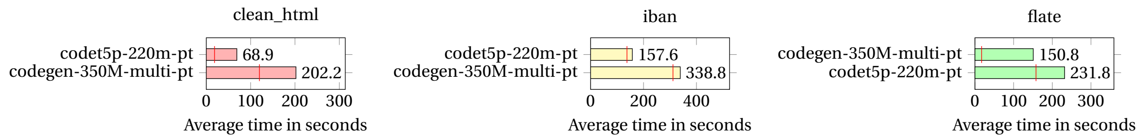


Figure 18: Average time to generate 10 seeds per test case

D Other evaluation results

Table 15: Reached bug location and average time until reached

Name	Combined		clean_html		iban		flate	
starcoderplus-pt	100%	51 ms	100%	19 ms	100%	111 ms	100%	23 ms
base-corp	100%	56 ms	100%	19 ms	100%	134 ms	100%	15 ms
gpt-3.5-turbo-ft	100%	72 ms	100%	19 ms	100%	175 ms	100%	22 ms
gpt-3.5-turbo-pt	100%	73 ms	100%	19 ms	100%	177 ms	100%	22 ms
starcoder-ft	100%	75 ms	100%	18 ms	100%	186 ms	100%	21 ms
codegen-16B-multi-ft	100%	78 ms	100%	19 ms	100%	193 ms	100%	21 ms
codegen-16B-multi-pt	100%	82 ms	100%	19 ms	100%	203 ms	100%	22 ms
instructcodet5p-16b-pt	100%	87 ms	100%	19 ms	100%	218 ms	100%	22 ms
codegen25-7b-multi-pt	100%	88 ms	100%	18 ms	100%	224 ms	100%	23 ms
codet5p-220m-pt	100%	91 ms	100%	19 ms	100%	232 ms	100%	21 ms
starcoderbase-pt	100%	93 ms	100%	19 ms	100%	238 ms	100%	22 ms
codet5p-16b-pt	100%	97 ms	100%	19 ms	100%	251 ms	100%	21 ms
gpt-4-pt	100%	98 ms	100%	18 ms	100%	253 ms	100%	21 ms
codet5p-770m-pt	100%	100 ms	100%	19 ms	100%	261 ms	100%	21 ms
codegen-350M-multi-pt	100%	102 ms	100%	18 ms	100%	266 ms	100%	23 ms
starcoder-pt	100%	106 ms	100%	18 ms	100%	278 ms	100%	21 ms
codegen25-7b-mono-ft	100%	108 ms	100%	19 ms	100%	285 ms	100%	21 ms
starcoderbase-ft	100%	112 ms	100%	18 ms	100%	296 ms	100%	21 ms
codegen-6B-multi-ft	100%	113 ms	100%	19 ms	100%	295 ms	100%	24 ms
codegen25-7b-multi-ft	100%	120 ms	100%	19 ms	100%	319 ms	100%	22 ms
codegen-6B-multi-pt	100%	123 ms	100%	18 ms	100%	329 ms	100%	22 ms
codegen-350M-multi-ft	100%	127 ms	100%	18 ms	100%	340 ms	100%	22 ms
codet5p-16b-ft	100%	127 ms	100%	18 ms	100%	339 ms	100%	23 ms
codet5p-770m-ft	100%	129 ms	100%	18 ms	100%	347 ms	100%	21 ms
base	100%	167 ms	100%	18 ms	100%	460 ms	100%	22 ms

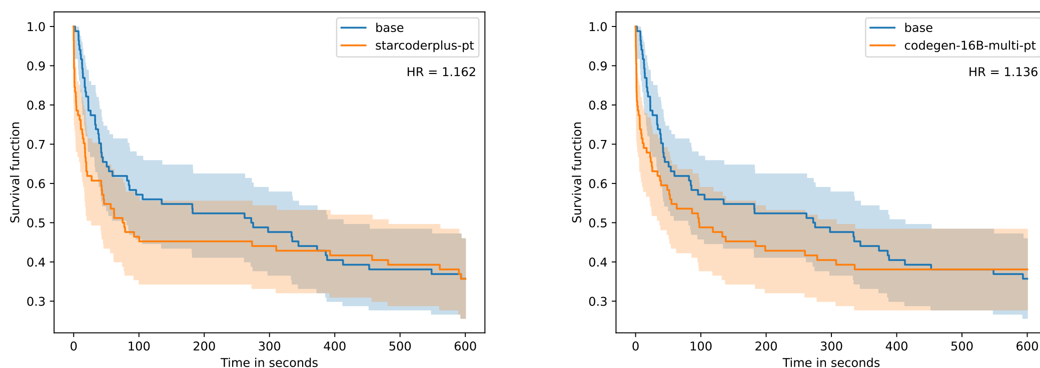


Figure 19: Extended Kaplan-Meier survival curves for best-performing methods within 30 seconds

D.1 starcoderplus-pt

Table 16: Model configuration results for starcoderplus-pt

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds*
temp_0.8.json	8	2	94 ms	146.09 s
top_p_0.99.json	8	1	32 ms	66.43 s
diverse_beam_search.json	8	1	32 ms	290.7 s
top_p_0.75.json	8	1	49 ms	134.18 s
temp_0.6.json	8	0	49 ms	171.78 s
temp_0.2.json	7	2	40 ms	195.66 s
top_p_0.50.json	7	2	59 ms	195.76 s

Table 17: Prompt-tune configuration results for starcoderplus-pt

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds*
text_multi.json	16	3	82 ms	188.09 s
text.json	13	3	33 ms	167.64 s
code.json	13	2	42 ms	167.69 s
code_multi.json	12	1	45 ms	162.63 s

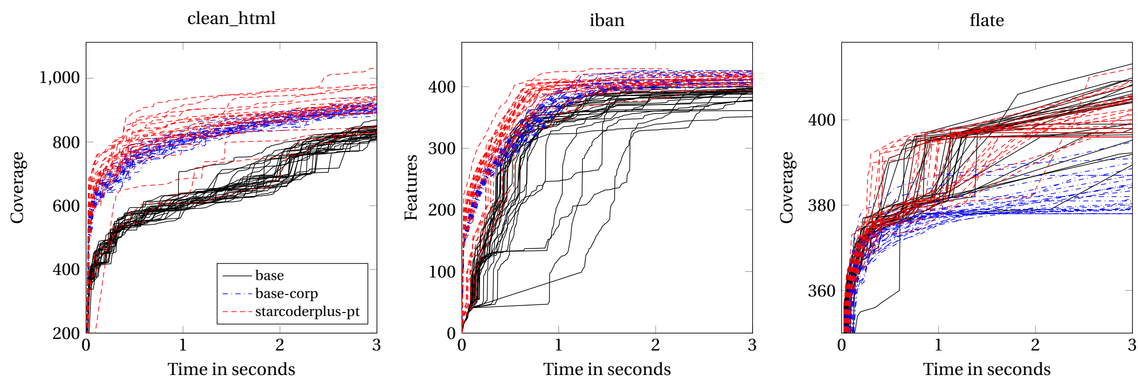


Figure 20: First three seconds of reached coverage

D.2 gpt-3.5-turbo-ft

Table 18: Model configuration results for gpt-3.5-turbo-ft

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds
top_p_0.99.json	9	2	73 ms	7.87 s
temp_0.8.json	8	3	81 ms	7.79 s
temp_0.6.json	8	2	70 ms	7.74 s
temp_0.2.json	8	1	89 ms	7.3 s
top_p_0.75.json	7	2	67 ms	8.19 s
top_p_0.50.json	5	1	50 ms	7.7 s
diverse_beam_search.json	1	1	71 ms	2.6 s

*See footnote C.2.

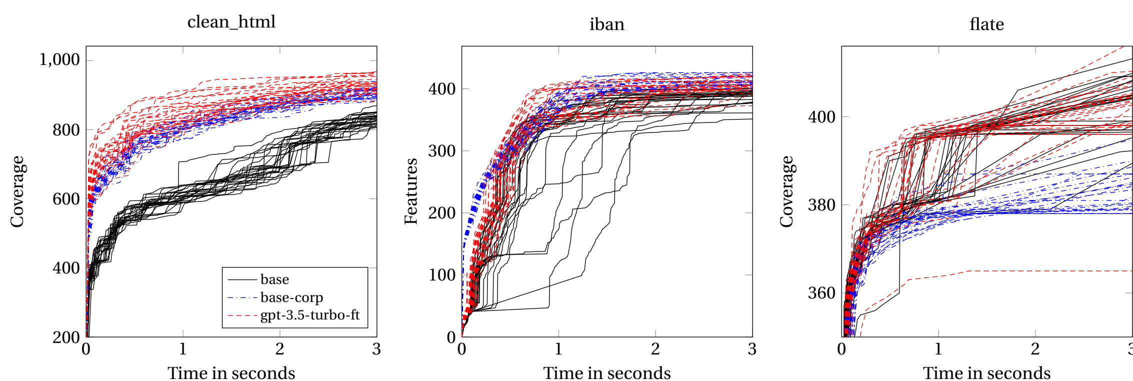


Figure 21: First three seconds of reached coverage

D.3 codegen25-7b-multi-ft

Table 19: Model configuration results for codegen25-7b-multi-ft

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds
temp_0.8.json	7	3	80 ms	34.44 s
temp_0.2.json	7	2	85 ms	33.13 s
top_p_0.99.json	7	2	97 ms	36.7 s
top_p_0.75.json	7	1	150 ms	36.9 s
top_p_0.50.json	6	1	111 ms	34.94 s
temp_0.6.json	6	1	132 ms	34.2 s
diverse_beam_search.json	2	0	394 ms	44.86 s

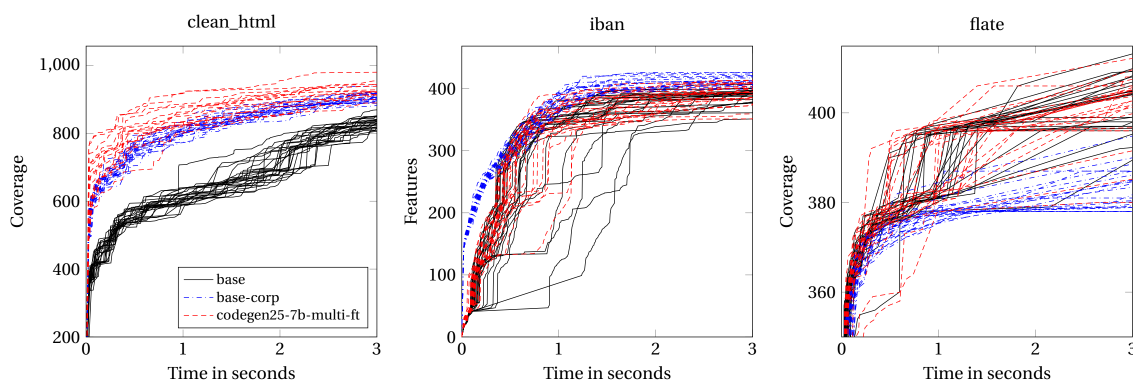


Figure 22: First three seconds of reached coverage

D.4 codegen-350M-multi-pt

Note that the generation times of codegen-350M-multi-pt are much longer compared to other models because the seeds were not generated on the A100 GPU.

Table 20: Model configuration results for codegen-350M-multi-pt

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds
temp_0.6.json	9	0	96 ms	134.56 s
top_p_0.99.json	8	1	71 ms	168.44 s
temp_0.2.json	8	1	129 ms	174.29 s
diverse_beam_search.json	8	0	102 ms	568.0 s
top_p_0.50.json	7	0	92 ms	223.69 s
temp_0.8.json	6	2	97 ms	128.0 s
top_p_0.75.json	5	1	129 ms	217.23 s

Table 21: Prompt-tune configuration results for codegen-350M-multi-pt

Config	Triggered	Unexpected	Avg Reached	Avg Gen 10 Seeds
text_multi.json	14	1	99 ms	275.37 s
code_multi.json	13	3	112 ms	264.49 s
text.json	13	1	100 ms	176.31 s
code.json	11	0	99 ms	206.24 s

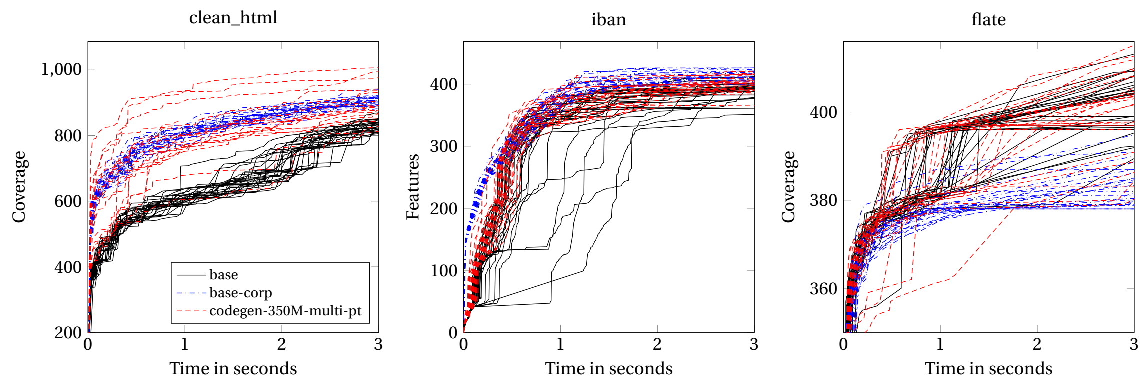


Figure 23: First three seconds of reached coverage