Thesis Proposal: Large Language Models for Software Evolution

Aidan Z.H. Yang

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Claire Le Goues, Chair
Ruben Martins
Vincent Hellendoorn

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: automated program repair, program verification, large language models

Abstract

Language models have improved by orders of magnitude with the recent emergence of Transformer-based Large Language Models (LLMs). LLMs have demonstrated their ability to generate "natural" code that is highly similar to code written by professional developers. However, the software engineering process involves much more than writing code: modern software evolves and requires continuous maintenance, such as debugging, or transpilation. For a LLM to assist in the software engineering process, it is important to build tools around a LLM to enable its ability to provide support for software evolution. In this thesis proposal, I propose mechanisms to improve the utility of LLMs for software evolution by using and combining LLMs with prior APR and program verification techniques. Specifically, I build LLM-based software engineering tools for fault localization, program repair, and program transpilation.

My thesis statement is: Software engineering is a process of evolution, including testing, debugging, and migration. LLMs have the potential for generating code that is largely correct. However, LLMs do not inherently have the ability to maintain evolving software. By using properties of software through program analysis and software verification, LLMs can assist in important steps of the software engineering process. In particular, LLMs can be a powerful tool for fault localization, program repair, and program language transpilation.

To support this statement, I propose the following contributions. In the preliminary work, which is already completed, I built a bidirectional fine-tuning technique that enables a previously left-to-right LLM to locate and rank faulty lines of code. I built the LLM-based fault localization technique without depending on preciously written tests, and so the tool can also detect run-time security vulnerabilities. I further study a LLM's ability for program repair by combining the LLM's intermediate entropy values with traditional program repair techniques. In particular, I used LLM entropy values to improve three stages of program repair: fault localization, patch testing efficiency, and plausible patch ranking. Finally, I created a tool for fully automated Rust function transpilation using LLMs and verification harnesses. To support my claims, I evaluated all my LLM-based software evolution tools on real world bugs, security vulnerabilities, and target transpilation repositories.

I propose to extend my study by further improving on the completed fault localizer mechanism with multi-task LLM instruction-tuning, and evaluating the model on real-world security vulnerabilities. Specifically, the completed fault localizer only detects faults on a line level, and does not take into account various nuances of security vulnerabilities. LLMs have the capability of training on multiple objectives, including both recognizing the vulnerable lines of code, and the explanation of exploits occurring from a type of vulnerability. Finally, I propose to extend the fault localizer to cover vulnerabilities that span across different functions, and files across an entire repository. To enable the detection of vulnerabilities from code across multiple files, I plan to encode dynamic taint tracking information as graph neural networks (GNNs) and train in parallel with LLMs.

Contents

1 Introduction

Software is constantly evolving, requiring constant care from software engineers to maintain software quality. The software evolution process includes bug finding, bug fixing, and sometimes an entire overhaul of the software project (e.g., a change in the programming language used). The problem of software evolution has motivated the development of a variety of techniques for Automatic Program Repair (APR) [45, 56, 58, 86, 89]. At a high level, dynamic APR approaches use test cases to define a defect to be repaired and functionality to retain, and to localize the defect to a smaller set of program lines. Specifically, fault localization (FL) [6, 49, 53, 60] approaches aim to automatically identify which program entities (like a line, statement, module, or file) are implicated in a particular bug. The goal is to assist programmers in fixing defects by pinpointing the places in the code base that should be modified to fix them. Broadly speaking, existing FL techniques combine or leverage static and dynamic program analysis information to compute a score corresponding to a program entity's probability of contributing to a particular bug.

Meanwhile, the recent advances in deep learning (DL) and AI have produced significant performance improvements over previous machine learning techniques for code generation [11, 16]. DL therefore affords promising opportunities for program repair [34, 42, 58, 86, 87] and fault localization [91]. The most effective DL models for both natural language and code related tasks are large language models (LLMs), such as Codex [17] and GPT-4 [15]. This class of models trains many billions of parameters with even more tokens of training data, which tends to yield highly flexible and powerful text generators. LLMs' utility for code generation and the fact that they are trained on an abundance of publicly-available code [17] both suggest that existing large-scale LLMs capture program source code in ways that can be leveraged for specialized development tasks. A key property of LLMs is that their performance improves consistently with the *scale* of their computational budget [36], which is itself a function of the model and training data size. However, LLMs are not immediately suited off-the-shelf for coding tasks that do not involve code generation, like fault localization.

To expand to the problem of automated program repair (APR), I believe that solely fine-tuning a LLM is not enough. The fundamental idea behind using an LM alone - even a hypothetically optimal one - for repair treats predictability as ultimately equivalent to correctness. This assumption is specious: LLMs adopt preferences based on a corpus with respect to training loss that rewards imitation. Beyond the fact that LLMs necessarily train on buggy code, LLMs generate and score text one token at the time. Given that, they may well prefer a subtly incorrect implementation spread across several readable lines over a correct but difficult-to-understand one-line solution, as the per-token probability of the former may be strictly lower than the latter. Judgement of code correctness requires substantially more context than an LLM has access to

including, but not limited to, test cases test behavior, and developer intent. Although some of this information could be provided as context, it will lie outside the training distribution. I argue that a careful combination of more traditional APR techniques with LLMs can produce the most effective results at all stages of APR.

As previously discussed, LLM-based approaches tend to produce code that is similar to their training data, and thus, if the model is trained on high quality, human written, code, the model will usually produce high quality, idiomatic code [92]. This gives the insight that LLMs can be powerful code transpilation tools (i.e., migrating from one programming language to another that directly compiles). a LLM can take a program in one language as input and attempt to output an equivalent program in the target language [73]. However, directly prompting a LLM comes with no formal guarantees that the resulting code will maintain input-output equivalence with the original [64, 66, 90]. Due to the unique nature of code transpilation, in which program equivalence can be represented by equivalence input-output pairings, I show that a LLM can produce verified transpilation when surrounded by test-harnesses and undergoing both testing and verification.

My key insight through building tools for fault localization, program repair, and program transpilation is that a LLM acts as a black box model for code generation, which can only be trusted when combined with non-ML based software engineering techniques.

1.1 Thesis Statement

Software engineering is an evolving process, including testing, debugging, and migration. LLMs have the potential for generating code that is largely correct. However, LLMs do not inherently have the ability to maintain evolving software. By using properties of software through program analysis, LLMs can assist in important steps of the software engineering process. In particular, LLMs can be a powerful tool for fault localization, program repair, and program language transpilation.

1.2 Contributions

This proposal contains a set of prototypes for addressing common software engineering tasks. My thesis will contribute in the following ways:

1. It introduces a fault localization specific LLM, while trained using light-weight bidirectional adapter layers.
2. It shows that the entropy values from various LLMs, when combined with prior APR tools, can improve upon all stages of APR as compared to either a LLM or traditional APR tool in separation.
3. It provides a set of techniques to safe-guard against LLM hallucinations for code-generation, which I evaluate on the task of transpiling to the Rust programming language.

1.3 Proposal Outline

In this proposal I first provide an overview of literature on the topics related to this proposal, and background on related topics (Chapter 2). In the following chapters Chapter 3, 4 and 5, I identify three research thrusts and describe the preliminary work done for each thrust as well as the proposed work. In Chapter 7, I propose a timeline for completing the ongoing research and finally in Chapter 8, I conclude this thesis proposal.

2 Review of Literature

The following sections give an overview of related work and background that inform the proposed work.

2.1 Related work

LLM for code Language models have been used for code generation, bug detection, and patch generation. Recent language models finetune on code as training data and can perform code completion [22, 27], and generate code based on natural language [72] with impressive results. Large Language Models (LLMs), such as Codex [16], GPT-Neo [11], and Llama-2 [79] have raised performance on these tasks by using more trainable parameters and training data. Ray et al. [70] study the relationship between bugginess and LLM-entropy. Ray et al. empirically showed that n-gram models trained over a large corpus of code will find buggy statements more surprising, as indicated by a high entropy score. Kolak et al. [40] revisit the question of naturalness (i.e., the human-readability) of patches in the era of large language models. Kolak et al. experimented with models ranging from [160M to 12B] parameters, and measured the similarly between LLM generated patches and developer written patches. Their results show that larger models tend to generate test-passing lines at a higher rate. Additionally, LLM generated patches tend to be more similar to the human-written patch as model size increases. Xia et al. [88] directly applied LLMs for APRs and found that LLMs can suggest multi-line fixes with higher accuracy than state of the art APR tools. Xia et al. [88] concluded that while LLMs can often perform localization and generation in one shot, it is significantly more cost-effective to use traditional FL first/instead, motivating my work.

Fault Localization Prior fault localization tools use test output information, code semantics, and naturalness of code to achieve a high degree of confidence on bug detection. Spectrum-based Fault Localization (SBFL) [5, 6] uses a ratio of passed and failed tests covering each line of code to calculate its suspiciousness score, in which a higher suspiciousness signifies a higher probability of being faulty. Recent advances in deep learning created a spur of research on using graph neural networks (GNNs) [75] for fault localization. GRACE [59], DeepFL [48], and DEAR [52] encode the code AST and test coverage as graph representations before training deep learning models for fault localization. TransferFL [62] combined semantic features of code and the transferred knowledge from open-source code data to improve the accuracy of prior deep learning fault localization tools.

Patch correctness Similarly to prior fault localization tools, prior patch disambiguation tools leverage test output information and code information (both code syntax and code semantics) for ranking or classifying patches. Qi et al. [69] analyzed the reported bugs of three generate-and-validate APR tools: GenProg [44], RSRepair [68], and AE [83] systems, to find that producing correct results on a validation test suite is not enough to ensure patch correctness. Smith et al. [76] performed an experiment that interrogates whether or not automatically generated patches are prone to overfitting to their test suite. Borrowing the concept of training and test sets from machine learning, they found that automated program repair (APR) typically used the same test-suite for both "training" (generating the patch), and "testing" (validation). Smith et al. found that both the coverage rate of the test-suite, as well as the assignment of test/train sets between the two suites, impact the degree of overfitting in repair. To counteract the overfitting problem, Ye et al. [95] proposed ODS (Overfitting Detection System), a novel system to statically classify overfitting patches. Xiong et al. [89] generated both execution traces of patched programs and new tests to assess the correctness of patches. Ghanbari et al. [29] used both the syntactic and semantic similarity between original code and proposed patch, and code coverage of passing tests to rank patches. Shibboleth [29] was able to rank the correct patch in Top-1 and Top-2 positions in 66% of their curated dataset. Tian et al. [78] proposed machine learning predictor with BERT transformer-based learned embeddings for patch classification. Tian et al. found that learned embeddings of code fragments with BERT [23], CC2Vec [31], and Doc2Vec [43] yield similarity scores that, given a buggy code, substantially differ between correctly-patched code and incorrectly-patched one. Yang et al. [94] found that state-of-the-art learning-based techniques suffered from the dataset overfitting problem, and that naturalness-based techniques outperformed traditional static techniques, in particular Patch-Sim [89].

Equivalence verification Equivalence verification has been studied in settings where two copies of should-be-equivalent artifacts are available: compiler optimizations [18, 19] and program verification [7, 41]. Our work uses equivalence verification for program language translation.

Rust transpilers Prior Rust transpilers convert C/C++ to Rust. C2Rust [1] automatically converts large-scale C programs to Rust while preserving C semantics. Citrus [3] and Bindgen [2] both generate Rust FFI bindings from C libraries, and produce Rust code without preserving C semantics. Bosamiya et al. [13] embedded WebAssembly (Wasm) semantics in safe Rust code for the Rust compiler to emit safe and executable Rust code. Bosamiya et al. [13] implemented all stages of their tool in safe Rust, and no stage of compiler needs to be further verified or trusted to achieve safety. Existing tools for making unsafe Rust safer focus on converting raw pointers to safe references. Emre et al. [24] localized unsafe Rust code from C2Rust and converted unsafe Rust to safe Rust by extracting type and borrow-checker results from the `rustc` compiler. Zhang et al. [96] converts unsafe Rust from C2Rust to safe Rust by computing an ownership scheme for a given Rust program, which decides which pointers in a program are owning or non-owning at particular locations. Zhang et al. [96] evaluates their tool CROWN on a benchmark suite of 20 programs, and achieve a median reduction rates for raw pointer uses of 62.1%. Ling et al. [54] removed non-essential "unsafe" keywords in Rust function signatures and refined the scopes within unsafe block scopes to safe functions using code structure trans-

formation. my work is the first to propose a general Rust transpiler that does not depend on the source language's memory management properties to produce safe Rust.

3 Fault localization

Debugging is an important step in the software maintenance process, and the first step in debugging is finding where the bug occurs in a software repository. As described in Chapter 1, fault localization (FL) approaches aim to automatically identify which program entities (like a line, statement, module, or file) are implicated in a particular bug. The goal is to assist programmers in fixing defects by pinpointing the places in the code base that should be modified to fix them. However, LLMs are not immediately suited off-the-shelf for coding tasks that do not involve code generation, like fault localization. State-of-the-art LLMs for code [12, 17, 65, 80] are trained to generate code in a left-to-right manner, with each token predicted from its preceding context. Models trained in this way are less suitable for token-level discriminative tasks, like line-level fault localization, because the representation for any given token is only conditioned on the context to the left.

My collaborators and I present a promising alternative: we train lightweight bidirectional *adapters*, themselves small models of the same architecture as the base LLM, on top of left-to-right language models. These adapters add relatively few parameters and can be trained effectively on small datasets of real bugs, such as *Defects4J* [35], without updating the underlying LLM. I demonstrate that the representations learned by pretrained left-to-right language models already contain a wealth of knowledge about the suspiciousness of lines of code, which increases strongly with the size of the LLM. I can leverage this power through my adapters to find bugs while requiring just a few hundred training samples for pretraining. my approach yields better fault localization performance than prior work while requiring significantly less data preprocessing overhead. Importantly, my approach does not use test cases at all, and therefore does not depend on test code quality for its performance.

3.1 LLMAO: Large Language Models for Fault Localization

In this section, I discuss the key ideas behind my language model enabled fault localization technique. Figure 3.1 shows an overview of *LLMAO*'s training setup. The input to *LLMAO* is a buggy program; its output is a list of *suspiciousness* scores corresponding to each code line's probability of being buggy – values close to 1 indicate that lines are likely defective. As shown in Figure 3.1, I first tokenize the input and then provide it to a pretrained left-to-right LLM. From this LLM, I obtain one (high-dimensional) *vector representation* per line, which I provide to a small bidirectional model that predicts bugginess probabilities for each line. I only train the final stage of this model; the LLM remains frozen and can be easily replaced with other powerful open-source models. Figure 3.2 shows a more detailed description of my language modeling

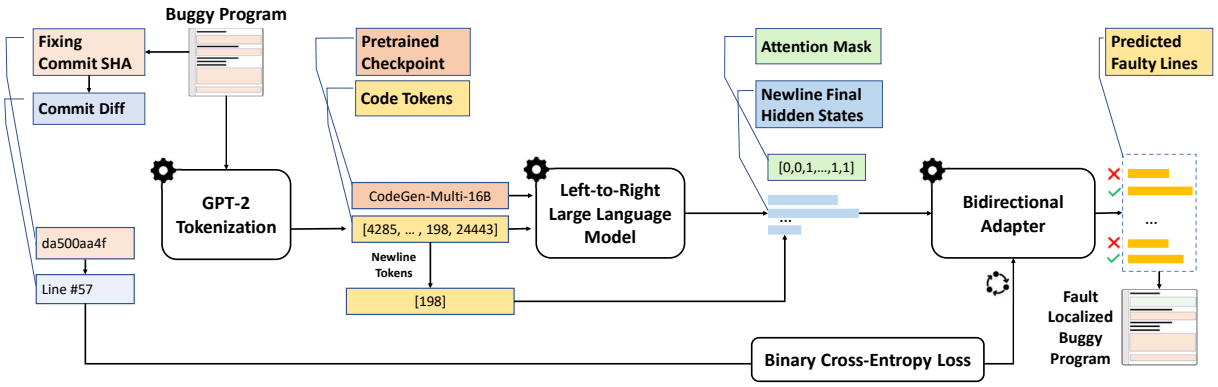

Figure 3.1: *LLMAO*'s architecture, which takes as input a buggy program and produces a list of suspiciousness scores for each code line

procedure, which I describe in detail in Section 3.1.2. In the following sections, I describe each component of *LLMAO*.

3.1.1 Left-to-right Language Models

Neural Language Models typically produce text in a left-to-right manner, producing each word given its prefix context. This both enables efficient training, as any document can be turned into a collection of as many training samples as there are tokens, and enables them to generate new text once trained. Virtually all modern language models are attention-based models that use the Transformer architecture [82]. In these models, each token exchanges information with all other tokens via a learned attention procedure. To efficiently train left-to-right Transformer models on an entire document in which each token is generated only from its prefix context thus involves "masking out" part of the attention matrix to prevent each token from attending to its suffix context (essentially, the future). Figure 3.2 (top) shows the causal attention mechanism used to train a left-to-right language model. Figure 3.2 describes a simplified Transformer model for both *CodeGen* and my bidirectional language model. Auto-regressive and left-to-right LMs [12, 17, 65, 80] use all previous tokens (i.e., tokens to the left) to predict the probability of the next token (i.e., tokens to the right). Left-to-right models are useful for program generation tasks, as shown in Figure 3.2. Specifically, the lower triangular part of the attention matrix remains unmasked (visualized as blue) while attention in the remaining part is masked out (white). This configuration allows each token to attend to itself and all past tokens, but prevents it from attending to future tokens.

my approach is compatible with any left-to-right language model, but is most effective when the underlying model is large and has been pretrained on a large volume of code data. At present, the *CodeGen* family of models [65] is most suitable for this role. These are a series of increasingly large left-to-right language models trained for program synthesis in three stages:

1. Each model is first trained on the natural language dataset ThePile, an 825.18 GiB mostly English language text corpus collected by Gao et al. [28] for language modeling. 7.6% of the dataset is programming language data collected from GitHub repositories with more

than 100 stars.

2. The models are then further trained on data from the Google BigQuery GHArchive dataset, which consists of open-source code across multiple programming languages – C, C++, Go, Java, JavaScript, and Python.
3. Finally, the models are tuned on the BIGPYTHON dataset, which contains a large amount of Python data.

Checkpoints after each stage are released for every model size, ranging from 350M to 16B parameters. The 16B model outperforms the original Codex model [17] on a Python program synthesis task.

While language models are typically used to predict the next token, they can also return the "hidden" states from their final Transformer layer. When generating text, these states are converted to a next-token prediction via a simple linear transformation. As such, these states tend to represent the model's knowledge about the evolving context at each point in the file, making them intrinsically useful. As shown in Figure 3.2, I extract the final hidden states for each newline (NL) token in each training sample from CodeGen to produce a condensed sequence representation in which each token represents one line. I pair these with their corresponding location (i.e., line #5 of a 50 line file) and save these to disk.

To train my model, I load these encoded lines in batches, where I retrieve samples of up to 128 contiguous newline states at a time. I choose this number because the *CodeGen* model can consume a maximum of 2,048 tokens as its input size; inputs with 128 lines almost always fit this token budget. Samples with fewer lines are padded, along with the label vector, to obtain a uniform length. Padding entries are ignored in the loss computation. When files contain more than 128 lines, I sample a series of 128 line windows that cover each faulty line in the file. Specifically, I repeatedly create a sample with up to 128 lines starting from a random offset before the immediate next faulty lines that is not yet covered by a previous segment. I then mark all faulty lines in this segment as covered and repeat until all lines are covered by at least one segment. I choose random starting offsets to ensure that the faulty lines within the split code lines are not consistently at the same indices (e.g., right at the start or in the middle), which would cause my model to memorize certain index locations as faulty lines. This enables my technique to handle inputs longer than 2048 tokens.

3.1.2 Bidirectional Adapter

While left-to-right language models extract rich representations per token, they are ill-suited for fault prediction because the representation of each given token only reflects knowledge of its leftward context. One solution might predict buggy lines based on the final hidden state, reflecting the model's knowledge after the entire file has been processed, but this creates a bottleneck where that state must store information from each line in the entire file. This bottlenecking phenomenon [9] is precisely why the NLP field moved away from Recurrent Neural Networks, which represent sequences with a single hidden state, and towards attention-based models, which preserve and use the state of each token [82].

I postulate that I can leverage these rich learned representations at each token by training just a few more Transformer layers that allow the model to exchange information between represen-

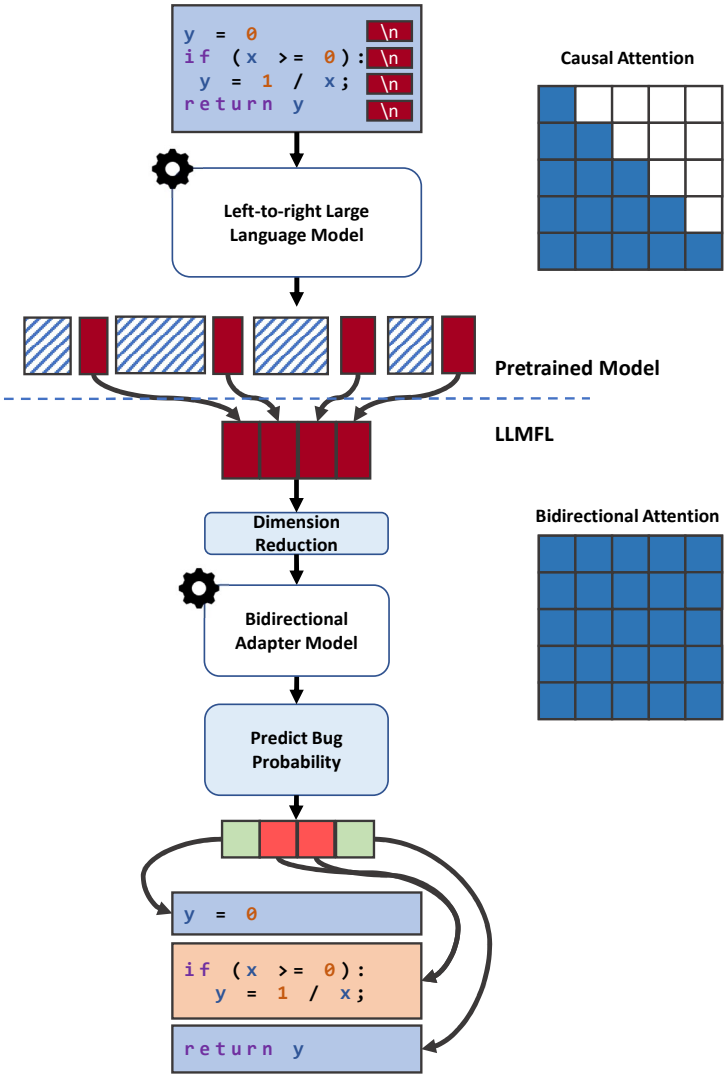

Figure 3.2: Attention masking procedure of *LLMAO*

tations of later and earlier lines, thereby generating a new, bidirectionally aware representation for each line of code. I can do so by removing the causal attention mask that normally prevents the exchange of information with "future" tokens in my added layers. In my case, I assume that the entire file has already been written, so this constraint is unnecessary. This yields a *bidirectional* encoder. As shown in Figure 3.2, the attention masking matrix for the bidirectional model allows all tokens in the sequence to attend to each other (visualized in blue). I thus train a bidirectional adapter consisting of a configurable number of Transformer layers, following the standard Transformer encoder architecture [82]. Concretely, my approach involves five steps, visualized in Figure 3.2:

Step 1: I start with a series of code tokens $C = [c_0, c_1, \ldots, c_N]$. I query a causally pretrained Transformer $\mathcal{T}_{PT}$ to transform these into a representational "states" $S \in \mathbb{R}^{N \times D}$, where D represents the pretrained model's dimensionality. This step takes place "offline", as I do not tune the

pretrained model.

Step 2: I extract the representations of each newline token to obtain state per line in the original program: $S_{NL} \in \mathbb{R}^{M \times D} = S[c_i = \backslash\text{n}]$, where M is the number of original newlines and typically $M \ll N$. I conjecture that these tokens' states reasonably accurately capture the information of their line in the file's prefix context.

Step 3: The dimension of the pretrained model's states, D, ranges up to 6,144 for the CodeGen models I built on. I use a significantly smaller dimension $d \ll D$ for my adapter layers, because they are trained on limited data. I first reduce the dimension of S_{NL} to $R_{NL} \in \mathbb{R}^{M \times d} = S_{NL}\, W_d$ where $W_d \in \mathbb{R}^{D \times d}$ is a learnable weight, equivalent to a fully connected layer. I experiment with dimensions $d \in \{256, 512, 1024\}$

Step 4: I then train an n-layer bidirectional Transformer adapter $\mathcal{T}_A$ with the same internal dimension d. This gives us the final representation of each newline token $A_{NL} \in \mathbb{R}^{M \times d}$, which aims to capture their role in the bidirectional context. I set the number of Transformer layers to $n = 2$.

Step 5: I transform each newline token's representation to a single value ranging from 0 to 1 via a sigmoid-activated dense projection $B = \sigma(R_{NL}\, W_b)$ where $W_b \in \mathbb{R}^{d \times 1}$. The resulting predictions per newline token can be seen as probability estimates of each line being buggy according to the model. These are compared against the ground-truth labels $T \in \{0, 1\}^M$ using the binary cross-entropy loss $\mathcal{L}_{CE} = T \ln B + (1 - T) \ln (1 - B)$. This loss is backpropagated through all layers up to, but not including, those in the pretrained network to obtain gradients. Given these gradients averaged across a sufficiently large minibatch of samples, the model states are updated to make its predictions more likely to agree with the training labels.

3.2 Evaluation and Results

3.2.1 Dataset

my work investigates the effectiveness of LLMs in the setting of fault detection. To determine how well my proposed technique can perform on real world faults, I select fmy datasets with source code and corresponding labeled fault lines.

- ***Defects4J* V1.2.0** : A Java benchmark dataset with 395 bugs from 6 Java projects [35]. I use V1.2.0 for most of my benchmarks instead of the latest version (V2.0.0) to compare on the same dataset as most prior FL techniques.
- ***Defects4J* V2.0.0**: A Java benchmark dataset with additional bugs over *Defects4J* V1.2.0 [35]. To show that my approach can generalize to faults from unseen projects, I further evaluate my tool as trained on *Defects4J* V1.2.0 on 226 new bugs from the newest *Defects4J* version (from projects totaling 165k more lines of code). I exclude the first 45 bugs in Jsoup and all in Gson/Jacksoncore because of trouble reproducing them (as seen in prior work [60]).
- ***BugsInPy***: a Python benchmark with 493 bugs from 17 different projects [84].
- ***Devign*:** a C benchmark with 5,260 from two open-source projects [97]. The original Devign dataset contains 15,512 security vulnerabilities from fmy different projects [97].

However, the authors of Devign only released a partial dataset available online.
All datasets include fixing commits that correspond to each fault. I identify faulty statements as those that are changed in the git diff associated with each commit, following prior approaches [50, 63, 71]. I then track line numbers of changed statements as training labels.

3.2.2 Baselines

LLMAO takes as input source code, and outputs a ranked list of probabilities corresponding to how likely a code line is buggy. To the best of my knowledge, no existing FL approaches take as input only the source code as natural language. However, I compare against existing FL approaches that take as input both source code and test code to observe if a LLM-enabled FL technique can produce comparable results without the dependence on tests or test coverage information.

my baselines are recent, state-of-the-art statement-level MLFL approaches: DeepFL [49], DeepRL4FL [51], and TRANSFER-FL [63]. DeepFL, and DeepRL4FL are MLFL techniques that take the test coverage information as model input. TRANSFER-FL builds on previous test-based MLFL approaches with pretrained information from open-source Java programs. I also include Ochiai [5], the best-performing SBFL approach. I use the prior techniques' replication packages to compute Top-N scores, including their handling of tied ranks (if any); I follow DeepFL's approach for accounting for tied ranks for Ochiai.

my tool produces a fault probability score for each line of a code file (i.e., statement level fault localization). Previous approaches output a ranked list of either suspicious statements or suspicious methods. In particular, DeepFL [49] is trained at the method level, i.e., predicting which methods are defective.

To compare, I follow other prior work and use DeepFL's spectrum and mutation-based features that are applicable to detecting faulty statements. DeepRL4FL, and TRANSFER-FL perform statement-level fault localization by default, similar to *LLMAO*. Since the repository and processed dataset for DeepRL4FL are not publicly available, I directly cite the experimental results reported in their paper [51]. For each of the other compared techniques, I run their tool for a total time of 30 minutes, which is comparable to my tool's training time for 300 epochs.

3.2.3 Validation

For each of my three datasets, I perform a 10-fold cross validation on the entire dataset. Specifically, I shuffle the dataset and train 10 models with 90% of the training set each, holding out the remaining 10% for validation, so that each sample in the dataset is held out exactly once. This is by contrast with some prior evaluations that in their default settings, validate tools within individual projects (using one bug in a given *Defects4J* project for validation and training on other bugs in that same project) [49, 51, 60, 63]. An effective and robust FL tool using machine learning or language models should be able to predict faulty locations while trained on code from different projects.

Training FL models on a particular project may produce over-fitting to a particular project and reduces applicability, requiring a relatively rich project and bug history before a technique can be used. I therefore believe that my 10-fold validation approach is more generalizable for

Table 3.1: Hyperparameters used for model training, both for the model trained from scratch and the three models trained on top of the various *CodeGen* models

Hyperparameter	**From Scratch**	**350M**	**6B**	**16B**
Max learning rate	5e-6	1e-4	7e-6	4e-6
Min learning rate	1e-8	1e-7	1e-7	1e-7
Model dimension	256	1024	4096	6144
Layers	8	2	2	2
Batch size	64	32	32	32
Epochs	2000	300	300	300

training models on larger code datasets. As is done in some prior evaluations [49, 51], I also *separately* evaluate the degree to which my model trained on one set of projects generalizes to a set of projects not seen in training (without retraining for those new projects).

I also deploy an early-stopping mechanism for each of my training runs. I checkpoint and record the epoch with the single highest average precision and recall score on the held-out validation dataset after every epoch. Once these scores stop improving for sufficiently many epochs (i.e., around 300 for all my model configurations), I stop training and use the best-performing checkpoint to calculate the Top-N metrics against the ground-truth labels.

3.2.4 Evaluation Metrics

I use the following evaluation metrics:

Top-N. Top-N measures the number of faults with at least one faulty element located within the first N positions (N=1, 3, 5). Developers only examine a small amount of the most-likely buggy elements within a ranked list [67], with particular attention paid to the top-5 elements [39]. To compare against state-of-the-art techniques, I adopt Top-N following prior work [49, 53, 60].

AUC of the model's ROC Curve. Although most developers inspect only top-5 elements in a given list, I also aim to measure how overall prediction compares against the ground truth. A Receiver Operating Characteristic (ROC) curve shows the performance of one classification model at all thresholds. It can be used to evaluate the overall model strength for making precise and accurate predictions. The area under an ROC curve (AUC) measures the usefulness of a test. AUC is a number between 0 and 1; higher is better. I measure the AUC at each of my model's top performing points in time, averaging precision and recall. I choose AUC to observe the prediction strength of my models at their peak performance.

3.2.5 Ablations

I conduct an ablation analysis to evaluate the impact of different components on the performance of my model (RQ3). I run five variants of my proposed technique for the *Defects4J* V1.2.0 dataset. I first evaluate *LLMAO* pretrained on *CodeGen*, and *LLMAO* without any pretraining

Table 3.2: *LLMAO* performance on 395 bugs from *Defects4J* V1.2.0, compared to prior techniques (top); on 226 additional bugs from *Defects4J* V2.0.0 (middle); and with ablation (bottom, again on defects from *Defects4J* V1.2.0)

FL type	Technique	Top-1	Top-3	Top-5
SBFL	Ochiai	19 (4.8%)	65 (16.5%)	99 (25.1%)
MLFL	DeepFL	57 (14.4%)	95 (24.1%)	135 (34.2%)
	DeepRL4FL	71 (18.0%)	128 (32.4%)	142 (35.9%)
	TRANSFER-FL	86 (21.8%)	135 (34.2%)	160 (40.5%)
LMFL	*LLMAO* with *CodeGen*-350M	82 (20.8%)	106 (26.8%)	126 (31.9%)
	LLMAO with *CodeGen*-6B	85 (21.5%)	115 (29.1%)	160 (40.5%)
	LLMAO with *CodeGen*-16B	**88 (22.3%)**	**149 (37.7%)**	**183 (46.3%)**
LMFL, new projects	*LLMAO* with *CodeGen*-16B	72 (31.9%)	93 (41.2%)	123(54.4%)
Ablation	$-pretraining$ (6 layers, trained from scratch)	5 (1.3%)	24 (6.2%)	30 (7.6%)
	$-bidirectional\ adapter$ (predict directly from *CodeGen*-16B)	10 (2.6%)	60 (15.2%)	85 (21.5%)

to evaluate the impact of the pretrained large language model's final hidden states. For the pretrained models, I checkpoint with three different *CodeGen* sizes (i.e., 350 million, 6 billion, and 16 billion parameters) to evaluate the impact of the pretrained model's size on finetuning. I also train a version of my model without bidirectional layers, using only the *CodeGen* auto-regressive attention mechanisms for fault localization. I aim to determine if left-to-right LLMs can detect faults directly, without any customization for code understanding.

3.2.6 Hyperparameters

Table 3.1 shows the hyperparameters used in training all my models. I reduced the learning rates until both the training and validation loss converged in a stable manner. Following the established practice in language model training [32], I use a learning rate warm-up of 1000 steps and a cosine learning rate decay until a global minimum learning rate of 1e-7 across 20k steps. Each model is trained on a single GPU. For the *CodeGen* pretrained models, I use a uniform batch size of 32 and perform gradient accumulation to ensure every batch of my data fits on a single GPU. For a fair comparison of *LLMAO*'s components (RQ3), I use the same number of training epochs (300) for all pretrain sizes and projected dimensions. However, the non-pretrained bidirectional model requires a much longer training time (some 2,000 epochs) for the validation loss to converge.

I train all configurations of my model on a uniform dimension of 512, which is projected

down from the various *CodeGen* models' hidden state dimensions (i.e., 1024, 4096, and 6144). I use a 8 attention heads for all my models.

3.2.7 Results

RQ1: How does *LLMAO* compare with prior DL-based FL tools? Table 3.2 (top) details experimental results showing how my tool compares against state-of-the-art FL techniques. The first 4 techniques are from prior approaches; I evaluate my *LLMAO* using three *CodeGen* pretrain sizes. The results show the Top-N (N $\in \{1, 3, 5\}$) score for each technique. Table 3.2 shows that *LLMAO* with the largest (16B) pretrained *CodeGen* size outperforms all the compared techniques. Even with smaller pretrain sizes (350M and 6B), *LLMAO* performs similarly to the top-performing prior methods.

Per Table 3.2, *LLMAO* with 16B *CodeGen* pretrain size detects 84 more faults within Top-5 than the top-performing SBFL technique, Ochiai (84.8% improvement). *LLMAO* detects 48 more faults within the Top-5 than the first proposed deep learning based FL technique DeepFL (35.6% improvement), and 23 more faults within the Top-5 than the latest state-of-the-art test-based MLFL approach TRANSFER-FL (14.4% improvement). For the Top-3 and Top-1 metric, *LLMAO* pretrained on the 16B *CodeGen* model can detect 14 more faults (10.4% improvement) and 2 more faults (2.3% improvement) than the state-of-the-art tool TRANSFER-FL. I observe that my LMFL technique improves particularly over prior FL techniques when more suspicious lines are inspected (i.e., higher Top-5 scores).

A Wilcoxon signed-rank test [85] indicates that the top-N values the difference between *LLMAO* with *CodeGen*-16B and prior techniques in terms of performance at the several top-N values is statistically significant (p-values ranging from 0.01 to 0.03).

When considering Top-1 scores, my approach is only slightly better than TRANSFER-FL, which performs roughly on-par with my *CodeGen*-6B model. However, note that prior techniques only achieve comparable results with my tool by requiring readily-available tests and test coverage as input. Writing tests and producing test coverage are time-consuming activities, and tests are not always available or useful when debugging. Furthermore, both DeepFL and TRANSFER-FL techniques include mutation-based fault localization information, which is very time-consuming to collect (i.e., hours of online collection time per bug [49]).

RQ1 Summary

LLMAO pretrained on the largest *CodeGen* size improves on the state-of-the-art by 14.4% on Top-5, without relying on test cases, program analysis, or even compilable code.

RQ2. How well does *LLMAO*'s performance generalize to new projects? I additionally evaluate *LLMAO* on bugs from the newer *Defects4J* V2.0.0, on projects that were not seen in pretraining (an additional 165K lines of code). The "LMFL, new projects" row in Table 3.2 shows that *LLMAO* with 16B *CodeGen* pretrain size detects 72/226 faults in top 1, 93/226 faults in top 3, and 123/226 faults in top 5.

Although I avoid strong statistical claims in this case study setting, these results are comparable to *LLMAO*'s performance on projects included in its training data, suggesting that it generalizes well. Several previously-published techniques are also evaluated for cross-project

Table 3.3: *LLMAO*'s Top-N Effectiveness on Different Datasets

Metric	***BugsInPy***	***Defects4J***	***Devign***
# lines	76,672	168,960	7,180,160
Top-1	51/493 (10.3%)	88/395 (22.3%)	1478/5260 (28.1%)
Top-3	59/493 (12.0%)	149/395 (37.7%)	2050/5260 (39.0%)
Top-5	75/493 (15.2%)	183/395 (46.3%)	3171/5260 (60.3%)

generalizability, in a variety of experimental settings. DeepFL and DeepRL4FL repeatedly train a model on N-1 projects and test it on a held-out project; in both cases, performance on the cross-project setting degrades compared to the within-project setting. GRACE [60] localizes to the method level (rather than the statement level); its cross-project evaluation also looks at defects from *Defects4J* V2.0.0. GRACE's performance also degrades slightly on new defects, though less than prior work. A key advantage of my approach is that *LLMAO* generalizes well to unseen projects *without retraining of any kind*. This argues for my technique's practicality both in terms of performance and time/compute requirements.

RQ2 Summary

LLMAO pretrained on the largest *CodeGen* size using data from *Defects4J* V1.2.0 performs well on bugs in unseen projects (not included in the training data), without additional training costs.

RQ3. How does each component of *LLMAO* impact its performance? The bottom two rows of Table 3.2 show the impact of pretrained models on *LLMAO*'s performance.

Without Pretraining I trained my bidirectional language model from scratch, using the same tokenizer as *CodeGen* for tokenizing the inputs. I replace *CodeGen*'s token-level representation with a learnable embedding for each token. I then pass these embeddings through 6 bidirectional Transformer layers (a typical minimum for Transformers) and predict the bugginess probability for states corresponding to newline tokens only (other tokens are embedded alongside these but ignored in the final layer). This model, trained on a sample size of 395 (i.e., total number of labeled *Defects4J* bugs) can achieve only a Top-1 of 5 (1.3%), Top-3 of 24 (6.2%), and Top-5 of 30 (7.6%). *LLMAO* without any pretraining performs significantly worse than *LLMAO* based on any size of *CodeGen*.

Without the Bidirectional Adapter I train a single linear projection from *CodeGen*-16B's final hidden states to a bugginess score for each line, thus omitting the bidirectional attention adapter layers. This approach performs better than *LLMAO* trained from scratch, with a Top-1 of 10 (2.6%), Top-3 of 60 (15.2%), and Top-5 of 85 (21.5%). This highlights how much program understanding a left-to-right LLM trained on a large corpus of code encodes in its learned representations. Although left-to-right models are not targeted at text-understanding, a LLM that can generate code given a natural language prompt can evidently learn to understand faults to a similar level of top performing SBFL approaches. Given enough fine-tune training on top of the previous task of code generation, CodeGen-16B without any further configuration is able to detect 85 *Defects4J* bugs (21.5%), which is only 14% worse than the top performing SBFL

model Ochiai. However, using *CodeGen*-16B for fault localization directly still performs significantly lower than all *LLMAO* models with bidirectional adapter layers. I perform an additional Wilcoxon signed-rank test [85] to observe that the top-N values of *LLMAO* with *CodeGen*-16B yields significantly better results than *LLMAO* without pretraining and without the bidirectional adapter at $\alpha = 0.05$ (p-values of 0.008 and 0.02).

Underlying LLMs Comparing my tool on different pretrained *CodeGen* sizes, I see an improvement in fault detection as the underlying model grows. *LLMAO* pretrained on *CodeGen*-350M improves upon *LLMAO* without the bidirectional adapter layers by 72 on Top-1. *LLMAO* pretrained on *CodeGen*-6B can detect 3 more faults on Top-1 than *CodeGen*-350M, and *LLMAO* pretrained on *CodeGen*-16B can find an additional 3 compared to *CodeGen*-6B. At higher Top-N targets, the performance improves more steeply with the size of the underlying model. For instance, *LLMAO* fine-tuned on *CodeGen*-350M detects 96 more faults than without pretraining, while fine-tuning on top of *CodeGen*-16B uncovers another 153.

RQ3 Summary

Although left-to-right language models can directly localize some faults, adding the bidirectional adapter layers is crucial for achieving state-of-the-art fault localization. Furthermore, I show that my tool using the largest pretrained LLM (i.e., *CodeGen* 16B) significantly outperforms all other variations of my model.

RQ4. How generalizable is *LLMAO* to other languages and domains? To evaluate my proposed technique on different languages and domains, I run all three pretrain sizes of my tool on the *BugsInPy* [84] dataset for localizing Python bugs, and the *Devign* [97] dataset for localizing C security vulnerabilities. I believe that measuring my tool on two other languages and one other defect domain can evaluate the effectiveness of modeling code defects as specific behaviors in natural language.

I observe from Table 3.3 that *LLMAO* can localize faulty statements with Top-1 of 10.3% on *BugsInPy*, and 28.1% for *Devign*. I observe that the performance of *LLMAO* improves as the size of the training dataset increases. Although *Defects4J* has fewer bugs than *BugsInPy*, I find that in total, *Defects4J* has 53% more code lines combined from all code files than the *BugsInPy* dataset. Since my approach considers source code as natural language, a larger database of code lines gives my models more training data. In particular, my largest dataset *Devign* with over 7 million lines of code achieves a Top-5 of 60.3% (i.e., 60.3% of my model's top-5 suspicious lines have at least one line that is an actual vulnerability).

Figures 3.3, 3.4 and 3.5 show the ROC curve for each of my trained models compared to the completely random curve (i.e., AUC=0.5). A ROC curve shows the performance of my model at all classification thresholds. The completely random curve has the true positive rate equal to the false positive rate at every classification threshold. I plot the ROC for my model trained on *Defects4J*, *BugsInPy*, and *Devign* after 300 epochs without any pretraining (i.e., the Transformer ROC curve), *CodeGen*-350M pretraining, *CodeGen*-6B pretraining, and finally *CodeGen*-16B pretraining.

I observe a clear improvement on the AUC as I use the *CodeGen* final hidden states for training, and the AUC continues to improve as I use larger *CodeGen* models. In particular, the AUC for Figure 3.3 yields 0.539 on *Defects4J* trained from scratch, 0.573 on *Defects4J* trained

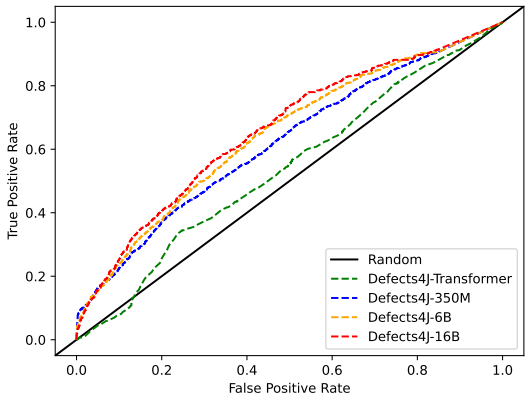

Figure 3.3: ROC curves on the *Defects4J* dataset

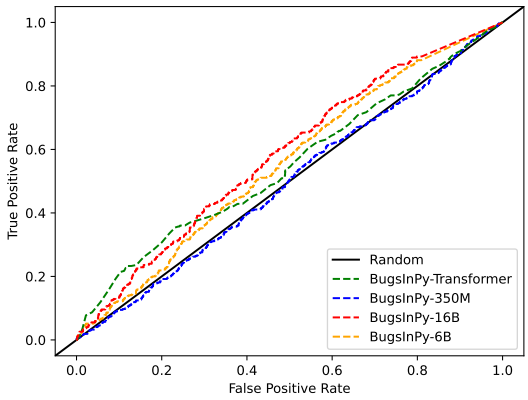

Figure 3.4: ROC curves on the *BugsInPy* dataset

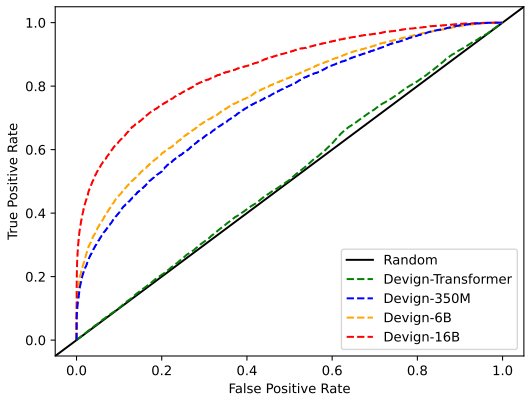

Figure 3.5: ROC curves on the *Devign* dataset

Figure 3.6: ROC curves on the completely random prediction, my model without any pretraining (Transformer), and pretrained on *CodeGen*-small (350M), *CodeGen*-medium (6B), and *CodeGen*-large (16B). Higher area under the curve (AUC) represents stronger predictive power.

from *CodeGen*-350M, 0.638 on *Defects4J* trained from *CodeGen*-6B, and 0.677 on *Defects4J* trained from *CodeGen*-16B. Figures 3.4 and 3.5 show a significant improvement in my model's predictive power as I use a larger dataset of code corpus. *LLMAO* with *CodeGen*-16B trained on my smallest dataset *BugsInPy* yields an AUC of 0.571, and *LLMAO* with *CodeGen*-16B trained on my largest dataset *Devign* yields an AUC 0.855. I observe that my model's predictive performance on *Devign* is better than my model's predictive performance on *BugsInPy* at all thresholds.

RQ4 Summary

LLMAO is more confident in its fault detection as the size of both training data and the pre-trained model scale up. *LLMAO* is also particularly effective for locating security bugs in C where test cases are not available.

3.3 Conclusion

As a first step in using LLMs for maintaining software evolution, I propose bidirectional adapter fine-tuning for localizing program defects, which include general logic defects as well as security vulnerabilities. I perform an empirical study on 395 real bugs from *Defects4J*, 493 bugs from *BugsinPy*, and 5,260 security vulnerabilities from *Devign*. The results show that my technique can outperform existing state-of-the-art deep learning based fault localization techniques without the dependency on any test cases.

4 Automated Program Repair

Automated Program Repair (APR) aims to help software engineers automatically patch software bugs. Although LLMs can immediately be used as code generators, it remains unclear how LLMs can be used as bug patchers. In this chapter, I argue that LLMs can be powerful tools for APR when combined with prior traditional APR techniques. When used correctly, LLMs can save both software testing time and manual patch selection time.

LLMs are trained on large volumes of code in which defects are relatively rare. Since their training objective encourages next-token prediction, well-trained language models tend to simultaneously perceive faulty code as unlikely (or "unnatural") and to produce code that is correct, as correct code is more "natural" [70]. The naturalness of code and unnaturalness of buggy code is now a well-established phenomenon [30, 70]. However, the bulk of prior research on this topic relied on relatively simple n-gram language models [14]. Compared to present-day LLMs, these models provided a very poor estimator of code predictability. The "unnaturalness" of buggy lines was therefore mainly useful as an explanatory metric, but showed limited utility for precisely localizing defects, let alone repairing programs. The recent advancement of much larger and more sophisticated LLMs have decreased model perplexities by multiple orders of magnitude. This makes them a much more accurate adjunct both for estimating naturalness and for fault localization or correct patch identification [88, 94].

My collaborators and I revisit the idea of naturalness for program repair. LLMs can only go so far on their own in reasoning about and fixing buggy code. It moreover motivates the use of traditional tools, which compress such information, as a complement to LLMs in repair, which has indeed shown promising recent results for the patch generation stage in particular [88] (acknowledging the risk of training data leakage in any such experiment [8]). I go beyond prior work by interrogating the role of entropy as a complement to traditional repair at every stage:

Plausible patch generation. APR approaches typically generate multiple potential code changes in search of *plausible* patches that cause the program to pass all tests. Executing tests (and to some extent, compiling programs to be tested) dominates repair time: the template-based approach *TBar* [56] spends about 2% of its total time creating patch templates, 6% generating patches from templates, and 92% running tests on generated patches. Regardless of the patch generation method (e.g., symbolic techniques [45, 61, 89], template instantiation [37, 56], or machine learning models [88]) repair *efficiency* is best approximated in terms of the number of patches that must be evaluated to find a (good) repair [57]. I show that entropy, when used to order candidate patches for evaluation, can improve the efficiency of generic template-based repair by 24 tested patches per bug, on average.

Patch correctness assessment. Plausible patches are not always correct, in that they can fail to

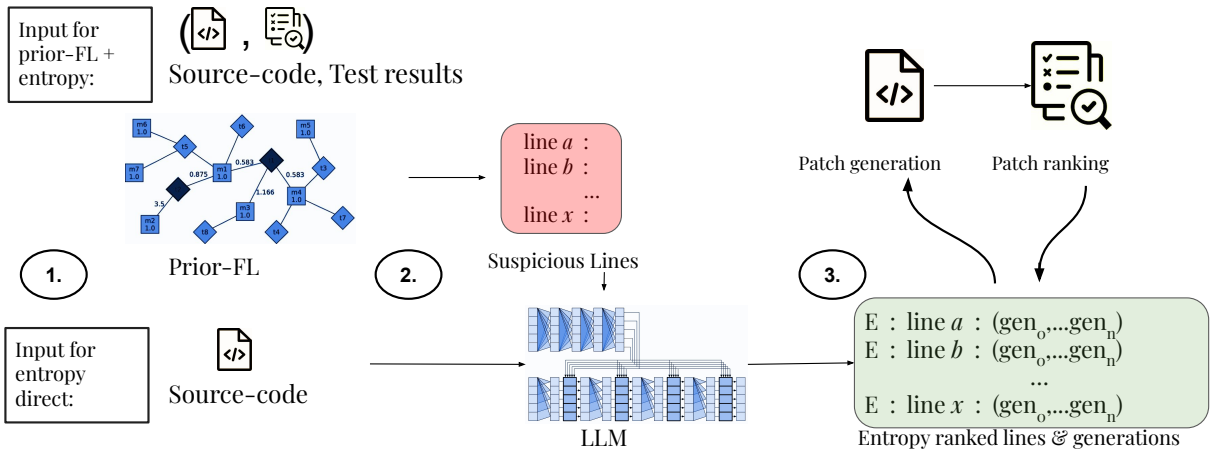

Figure 4.1: Fault localization pipeline using entropy. (1) I take a prior-FL suspicious score list, (2) query each code-line for entropy values, and (3) re-rank the list using LLM entropy scores.

generalize to the desired behavior beyond what is tested in the provided test suite [76]. Some recent work aims to address this in the form of a post-processing step that identifies (and filters) plausible-but-incorrect patches, typically by combining program analysis and machine learning [29, 78, 94]. However, techniques to date are typically trained on the same test suites used for patch generation, imposing a project-specific training burden (and an expensive one, when dynamic signals are required), and posing a significant risk of overfitting [76, 94]. I show that entropy can rank correct patches 49% more effectively (in Top-1) than state-of-the-art patch ranker Shibboleth [29], without using any project-specific training data.

4.1 LLM Entropy for APR

This section describes how I use entropy for evaluating patches; and my modifications to TBar to enable our study of improved patch efficiency. I calculate entropy with the following equation: $Entropy = -\sum_{i=1}^{n} \frac{log(p_{ti})}{n}$.

4.1.1 Entropy-Delta

To evaluate patch naturalness, which I use in both patch prioritization during generation/evaluation and patch correctness prediction, I introduce the concept of an "entropy-delta". Entropy delta describes how code replacement changes the naturalness of a block of code. Figure 4.2 and Figure 4.3 give examples for my usage of entropy-delta for assigning a ranking score for patches. Figure 4.2 shows the process of masking out a deleted line of code and querying the LLM for the change in entropy using that mask (i.e., the change in entropy without the original line). Figure 4.3 shows the process of querying the LLM for the change in entropy if the tokens of the original line of code is replaced with new tokens of patch code. If the patch is an insertion of a blank new line, I query the entropy-delta between the "newline" token and the original line

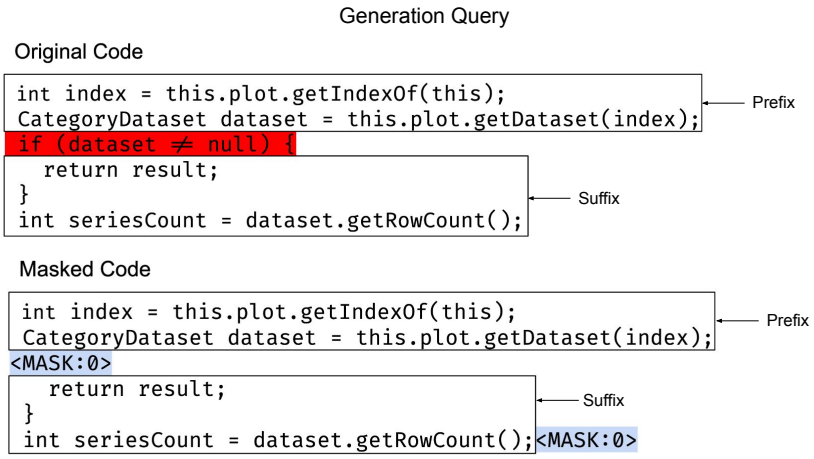

Figure 4.2: An example of entropy-delta query from a code-line deletion patch. The entropy-delta value of the deleted line is the difference between the original line and a blank line.

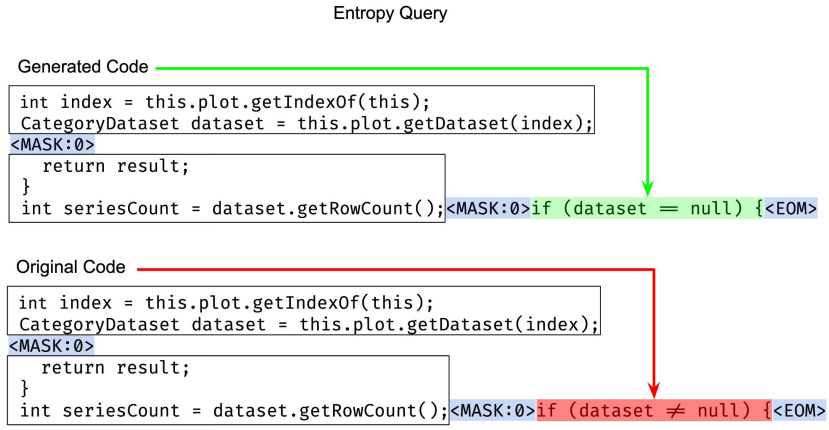

Figure 4.3: An example of entropy-delta query from a code-line replacement patch. The entropy-delta value of the replaced line is the difference between the original line and the replacement line.

of code. For the case of an insertion, I measure the entropy-delta between the new code line and the original blank line.

An entropy-delta is simply the change in entropy before and after a line in code is replaced. For instance, if the line's original entropy is 1.0, and the replacement line's entropy is 0.5, then the line has an entropy-delta of +0.5, as in, replacing that line lowered entropy by 0.5. A significant reduction in entropy (large, positive entropy-delta) means that the replacement code lowered the entropy, implying both that the original statement may have been buggy and that the patch is more natural for that region of code. A large, negative entropy-delta means that the replacement code increased entropy, meaning that the patch is less natural at that location. An entropy-delta of 0 means that the patch has the exact same naturalness as the original code.

4.1.2 Modified TBar

my patch efficiency experiments ask how entropy can speed up patch generation and evaluation. I evaluate it in context of TBar [56], the best-performing template-based program repair technique in the existing literature. I avoid using ML-based APR techniques (even though some may outperform TBar [52, 62, 88]) because my goal is a controlled evaluation of entropy without learned patterns from the test suite. Evaluating based on a technique that otherwise also relies on trained ML models fails to isolate the effect of entropy per se.

TBar is a template-based patch generation technique integrated with Defects4J V1.2. my experiments require several modifications to the codebase. First, I enable TBar to continue seeking patches after the first test-patching patch is found. Second, I enable TBar to generate patches, or evaluate them in a customized order (such as one provided by an entropy-delta ranking). my TBar extension also includes some refactoring for modifiability/extensibility, as well as a more accurate patch caching mechanism (caching the patched source code, rather than the patch alone). I provide the modified code with my replication package.

4.2 Evaluation and Results

In this section, I describe the models I use for entropy (Section 4.2.1), the bug and patch datasets considered (Section 4.2.2), as well as evaluation metrics (Section 4.2.3).

4.2.1 LLMs

I used inCoder [27], Starcoder [46], and Code-Llama2 [79]. The three LLMs were trained on open-source code and are capable of infilling with bidirectional context. The inCoder model [27] is a large-scale decoder-only Transformer model with 6.7 billion parameters. The model was trained on a dataset of code from public open-source repositories on GitHub and GitLab, as well as from StackOverflow. inCoder was primarily trained for code infilling, which involves the insertion of missing code snippets in existing code, using a causal-masked objective during training. However, its versatility enables it to be utilized in a variety of software engineering tasks, including automatic program repair. Starcoder and Llama-2 were trained with a similar autoregressive plus causal-mask objective as inCoder. Starcoder was trained with 15.5 billion

Table 4.1: Defects4J bugs with at least one patch passing tests (efficiency), and a developer fix (patch correctness).

	Defects4J V1.2 #bugs Patch efficiency		**Defects4J V2.0 #bugs** Patch correctness	
	Incl.	Total	Incl.	Total
Chart	11	26	19	26
Closure	19	133	64	174
Lang	14	65	35	64
Math	21	106	67	106
Mockito	3	38	1	38
Time	4	27	11	26
Total	72	395	197	434

parameters. Code-Llama2 have three versions available: 7B, 13B and 34B. I choose the 7B version as it is the closest in size to the other two models. Although the three LLMs were not specifically trained for repair, their large architectures and training objectives could imply that their entropy values on a particular region of code could suggest code naturalness. For all experiments, I set the LLM temperature to 0.5.

4.2.2 Dataset

I use the Defects4J [35] dataset as the basis of our experiments. Defects4J is a well-established set of documented historical bugs in Java programs with associated tests and developer patches. It is commonly used in APR, testing, and fault localization research. However, each research question requires a different subset of the data. Table 4.1 shows the number of bugs in each project that have at least one patch passing tests (for analyzing patch efficiency) and a developer fix (for analyzing patch correctness) along with plausible but incorrect patches. In total, I analyze 72 bugs from Defects4J V1.2 for patch efficiency and 197 bugs from Defects4J V2.0 for patch correctness.

I used Defects4J V1.2 for the fault localization and patch generation experiments. I do this because off-the-shelf TBar, as well as prior fault localization tools' replication packages, are all only compatible with Defects4J V1.2. The fault localization experiments consider all 395 bugs in Defects4J V1.2. I choose not to use Defects4J V2.0 for fault localization because prior tools' replication packages are only compatible with Defects4J V1.2.

For patch generation, the goal is to evaluate the degree to which entropy can improve repair efficiency; I therefore focus on the subset of Defects4J V1.2 bugs on which vanilla TBar succeeds at least once.

For patch correctness ranking, I use curated datasets from prior tools' replication packages directly, namely, Shibboleth [29] and Panther [78]. Shibboleth and Panther are both tools that leverage static and dynamic heuristics from both test and source code to rank and classify plausible patches, built on top of the updated Defects4J V2.0 dataset. We use a dataset of 1,290

Table 4.2: Entropy-delta ranking scores of 72 plausible patches generated by TBar per Defects4J project. The mean rank decrease is 24 and the median is 5.

Project	Improves ranking	Lowers ranking
Chart	11	2
Closure	15	4
Lang	11	2
Math	16	3
Mockito	1	0
Time	3	1
Total	60	12

plausible patches on Defects4J V2.0 curated by Ghanbari et al. [29]. For patch classification, I use a dataset of 2,147 plausible patches on Defects4J V2.0 curated by Tian et al. [78]. The patches from Tian et al. [78] were generated by seven different APR techniques. Each bug in the data set has one correct patch and several plausible (i.e., test passing) but incorrect ones. I calculate the change in entropy between the section of code in the original (buggy) file and the patched version. Note that both datasets only contain patches in projects Chart, Closure, Lang, Math, Mockito, and Time (6/17 of Defects4J V2.0's total projects), to compare with prior work built on Defects4J V1.2. Instead of the total number of bugs 835 in Defects4J V2.0, I only consider the 434 bugs in the 6 projects included by Shibboleth [29] and Panther [78] (shown in Table 4.1).

4.2.3 Metrics

Patch generation efficiency. I measure the effect of reranking generated potential patches in terms of the number of patch evaluations saved by doing so. Patch evaluations are established as a hardware- and program-independent measure for APR efficiency [57], and a proxy for compute time. **Patch correctness.** For patch classification tasks, I convert entropy-delta values into binary labels. We label patches with a positive entropy-delta as "more natural" (i.e., more likely to be correct), and patches with a negative entropy-delta "less natural" (i.e., less likely to be correct). To measure entropy's ability to isolate correct and incorrect patches, I use +recall and -recall. +Recall measures to what extent correct patches are identified, while -recall measures to what extent incorrect patches are filtered out. I use accuracy, precision, and F1 scores to assess classification effectiveness over the entire dataset.

4.2.4 Results

RQ1. Can entropy improve patch generation efficiency?

In this section, I discuss the observed relationship of entropy and test-passing patches. I use entropy from inCoder. I measure the impact of entropy-delta on patch generation efficiency with two methods: (1) measuring each successful (test-passing) patch's ranking as ranked by

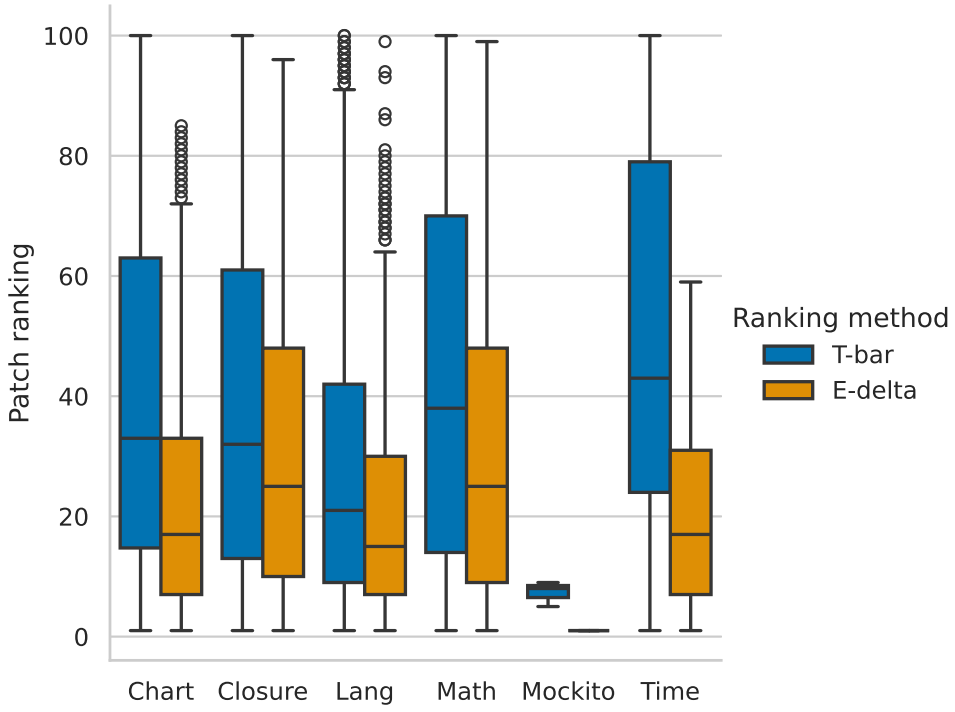

Figure 4.4: Entropy-delta and TBar ranking (lower is better) of test-passing patches on 72 Defects4J bugs.

original TBar and entropy-delta re-ranked TBar, and (2) incorporating entropy-delta into TBar and measuring the total number of patches generated to pass all tests.

I first configured TBar to generate only 100 patches per each Defects4J bug, assuming perfect fault localization. Of the TBar patches I generated, 72 passed all tests contained in their bugs' respective repositories (e.g., all tests written for project Chart). Finally, I calculated the entropy-delta score for each patch, and the test-passing patch's original ranking according to TBar. As seen in Table 4.2, entropy-delta improves 60 out of the 72 rankings as compared to TBar's original ranking. On average, we observed a mean rank decrease of 24, meaning that using entropy-delta to rank the generated TBar patches can reduce a mean of 24 full test iterations (i.e., each potential patch must run through all test cases in the repository before knowing if it is a plausible patch). Liu et al. [57] compared 16 APR techniques and found that TBar exhibits one of the highest number of patches generated, but also the highest rate of bug fixing across Defects4J. I posit that entropy-delta's efficiency improvement over TBar significantly boosts template-based APR's overall utility.

Figure 4.4 compares the TBar ranking and entropy-delta ranking. Each bar represents the rank of test-passing patches compared to all generated patches per Defects4J project. A lower rank signifies a more efficient repair process, as the repair process ends when a test-passing patch is found. As seen in Figure 4.4, TBar's original ranking for test-passing patches is higher than entropy-delta's ranking across all projects. Entropy-delta shows a higher disparity on ranking between test passing and test failing patches (i.e., a lower median rank for all test-passing

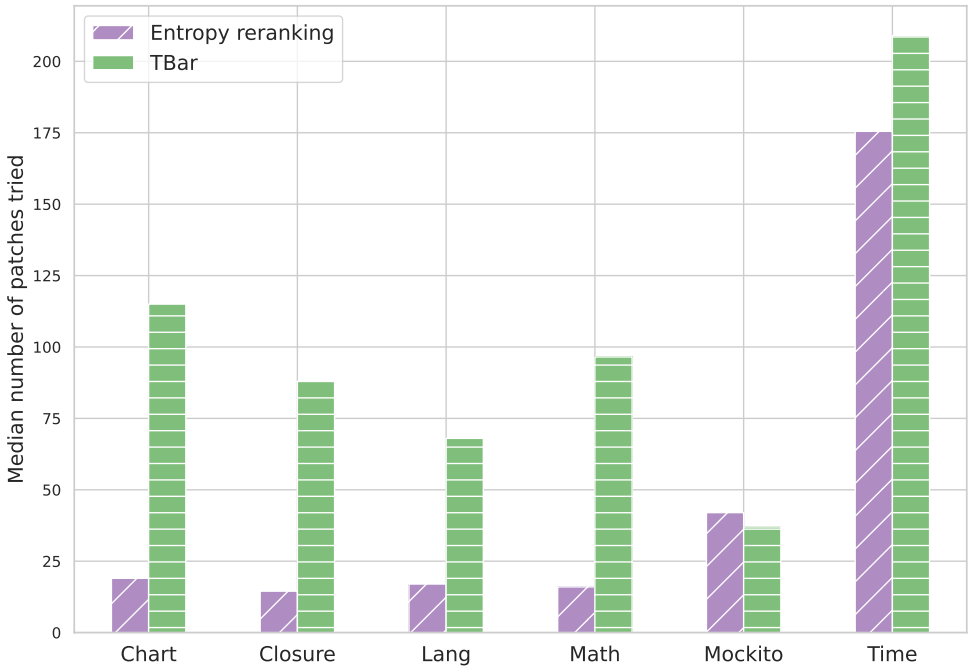

Figure 4.5: Median number of patches tested (lower is better) per project before successful patch using TBar original ranking and entropy-delta re-ranking of test-passing patches on 100 Defects4J bugs.

patches). In particular, patches from projects Chart and Time show the largest improvement from re-ranking patches with entropy-delta. Successful patches in Chart and Time typically require multi-line edits, and with a wider range of templates to choose from, entropy-delta can make a greater impact in reducing the number of patches tested.

I then configured TBar to use entropy-delta ranked patches directly, and measured the total number of patches required until a successful bug fix (i.e., passing all tests). Figure 4.5 shows the median number of patches tested per project before a successful patch using TBar original ranking and entropy-delta re-ranking. I observe that entropy-delta re-ranking reduces the median number of patches tested across all projects except for Mockito. Mockito has only three single line bugs that TBar can fix. With a smaller total number of patches to try on a single template (e.g., 11 total possible patches for Mockito-26), entropy-delta re-ranking does not have as large of an impact on APR efficiency.

Finally, I measure the actual correctness of TBar's first plausible patch, as ranked by Entropy. I find that, of the 72 plausible patches produced by TBar in my experiment, 19 are identical to the developer fix.

Table 4.3: Ranking results of 1290 plausible patches per Defects4J project using ranking methods SBFL, Shibboleth, and entropy-delta

Project	#Patches	#Correct	#Incorrect	Top-N	SBFL	Shibboleth	Entropy-delta
Chart	201	19	182	Top-1	3	11	10
				Top-2	6	14	14
Closure	269	64	205	Top-1	19	27	48
				Top-2	38	47	58
Lang	220	35	185	Top-1	1	14	20
				Top-2	12	22	27
Math	541	67	474	Top-1	10	27	39
				Top-2	30	38	55
Mockito	2	1	1	Top-1	0	1	1
				Top-2	1	1	1
Time	57	11	46	Top-1	3	8	9
				Top-2	5	5	10
Total	1290	197	1093	Top-1	36	85	**127**
				Top-2	92	130	**165**

RQ1 Summary

I show that entropy can be used to rank patches before going through the entire test-suite, thereby reducing the test overhead for template-based repair technique TBar by a mean of 24 patches tested. Entropy-delta can both reduce the median number of patches tried before finding a fix, and consistently rank test patching patches higher than test-failing patches without any dependency on the test-suite. Entropy-delta is most useful for bugs that require multi-line patches.

RQ2. How well does entropy-deltas identify correct patches?

I saw that entropy-delta can improve the efficiency of patch generation by reducing number of patches tested. However, it is important to note that a test-passing patch is not necessarily correct. To further explore the issue of correctness, I investigated the ability of entropy-deltas to distinguish between correct and incorrect patches, both of which are test-passing.

Patch ranking

I evaluate a dataset of 1,290 patches generated by 7 prior APR methods collected by Ghanbari et al. [29]. For each bug, the data set includes some number of plausible (i.e., test passing) patches, where exactly one is correct, and the rest are incorrect. I attempt to isolate the true correct patch from the incorrect patches. I then rank each patch according to its entropy-delta, querying the model for the entropy of the entire patch region before and after the replacement.

Following the approach by Shibboleth [29], I use Ochiai's SBFL for patch ranking as a base-

Table 4.4: Classification scores of 2,147 plausible patches on Defects4J projects using classification methods PATCH-SIM, Panther, and entropy-delta

Score	PATCH-SIM	Panther	Entropy-delta
Accuracy	0.388	0.730	0.735
Precision	0.245	0.760	0.900
+ Recall	0.711	0.757	0.760
- Recall	0.572	0.696	0.624
F1	0.377	0.750	0.824

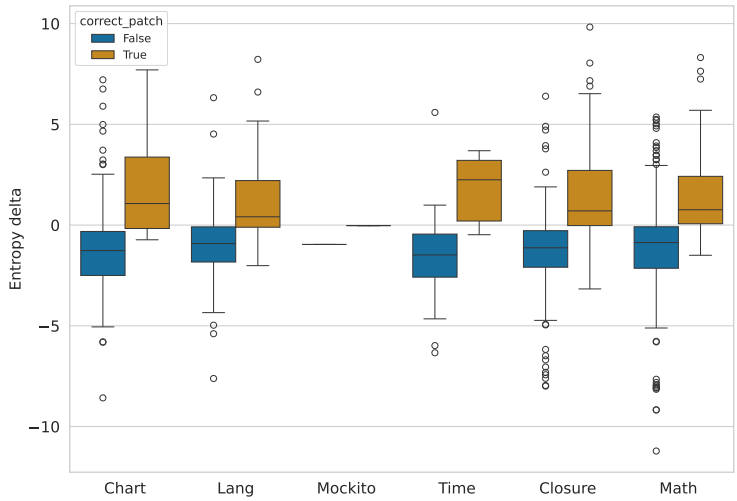

Figure 4.6: Entropy-delta across correct and incorrect patches on Defects4J projects. A higher entropy-delta signifies a less surprising patch to the LLM, and a lower entropy (sometimes negative) entropy-delta signifies a more surprising patch to the LLM.

line: ranking patches by the SBFL suspiciousness of the modified location.

Table 4.3 shows the Top-1 and Top-2 results of my approach on the labeled dataset of 1,290 patches. We see from Table 4.3 that entropy-delta outperforms both SBFL and Shibboleth [29] on Top-2 across all projects, and entropy-delta outperforms Shibboleth on Top-1 across all projects but Chart (10 Top-1 as compared to Shibboleth's 11 Top-1). Overall, I see that entropy-delta improves upon Shibboleth by 49% for Top-1, and 27% for Top-2.

The difference in entropy reduction between correct and plausible but incorrect patches is shown in greater detail in Figure 4.6. I see a clear difference in entropy-delta across correct and incorrect patches. In particular, the correct patches for all six projects have a median entropy-delta value of above 0, and the incorrect patches for all six projects have a median entropy-delta value of below 0. A correct patch tends to appear more natural to the LLM as compared to its original buggy line.

Patch classification

Table 4.4 shows my classification results on a labeled dataset of 2,147 plausible patches curated by Tian et al. [78] for classifying patches as correct or incorrect. Entropy-delta improves upon the accuracy score of PATCH-SIM [89] and Panther [78], but only slightly improves +recall score over both PATCH-SIM and Panther. For -recall, entropy-delta performs better than PATCH-SIM by 9%, but performs worse than Panther by 10%. Entropy-delta slightly improves accuracy over Panther by 0.6%, and 89% over PATCH-SIM. Entropy-delta improves precision over Panther by 18%, and PATCH-SIM by 267%. Finally, entropy-delta performs better than both PATCH-SIM and Panther on F1 score, by 118% and 10% respectively. As compared to the state-of-the-art, entropy improves classification performance on true positives more than true negatives.

Patch classification on machine learning APR

my analysis focused on comparing the degree of entropy reduction between true correct patches and plausible test-passing patches. As shown in Table 4.3, Table 4.4, and Figure 4.6, my results suggest that correct patches tend to lower entropy (i.e., increase naturalness) more than incorrect patches. Specifically, entropy-delta ranks 49% more correct patches in the Top-1 than the state-of-the-art patch ranker Shibboleth, and entropy-delta can classify correct patches with an 18% higher precision than the state-of-the-art patch classifier Panther. These findings suggest that entropy-deltas can be a valuable heuristic for distinguishing between correct and incorrect patches.

RQ2 Summary

The entropy-delta from a LLM distinguishes between correct and plausible (test-passing but incorrect) patches with higher precision and accuracy than state-of-the-art patch disambiguation tools.

4.3 Conclusion

After fault localization, a natural next step of software maintenance is program repair. In this chapter, I introduce the use of "unnaturalness" of code for automated program repair through the measurement of entropy generated by code-tuned LLMs. I coin the term entropy-delta, which measures the difference in entropy between a proposed code insert (i.e., a patch) and the original code. I use entropy-delta on untested patches to save an average of 24 test runs per bug for the template-based APR technique *TBar*. Entropy-delta can improve upon state-of-the-art patch ranking by 49% for Top-1, can classify plausible patches with a 18% higher precision, and is still effective for patches produced by modern ML-based tools. The results in this chapter indicate that LLMs can be a powerful addition to state-of-the-art APR tools without the dependency on tests, and the usage of LLM code generation. The reduction in both test suites and LLM code generation results in the reduction in model over-fitting and training data leakage.

5 Program Transpilation

A common component of software evolution is program translation, whether between platforms, software versions, or programming languages. Specifically, translating from one programming to another while directly compiling the target program is called program transpilation. One key difficulty in program transpilation is producing idiomatic code. In this chapter, I design, implement, and evaluate a LLM-based transpilation tool, and show that it is more idiomatic than prior non-LLM transpilers. Specifically, I use the Rust programming language as a case-study for my target transpiling language.

Rust is a programming language that combines memory safety and low-level control, providing C-like performance while guaranteeing the absence of undefined behaviors by default. Rust's growing popularity has prompted research on safe and correct transpiling of existing code-bases to Rust. Existing work falls into two categories: rule-based and large language model (LLM)-based. While rule-based approaches can theoretically produce *correct* transpilations that maintain input-output equivalence to the original, they often yield unidiomatic Rust code that uses unsafe subsets of the Rust language. On the other hand, while LLM-based approaches typically produce more idiomatic, maintainable, and safe code, they do not provide any guarantees about correctness.

In this work, I propose VERT (verified equivalent Rust transpilation with LLMs), a tool that can produce idiomatic Rust transpilations with formal guarantees of correctness. VERT's only requirement is that there is Web Assembly compiler for the source language, which is true for most major languages.

5.1 VERT: Verified Equivalent Rust Transpilation with LLMs

In this section, I describe the key ideas behind VERT, a universal Rust transpilation technique. Figure 5.1 gives an overview of VERT's entire pipeline. The technique takes as input a source program, and outputs a verified equivalent Rust program. As shown in Figure 5.1, I parse the program into separate functions during the cleaning phase, then split the pipeline into two paths. The first path outputs a LLM-generated Rust transpilation. The second path produces rWasm Rust code that is compiled directly from the original program through Wasm. Finally, I create a test harness based on the original program to verify equivalence of the two paths' outputs, and only after a successful verification I output a correct and maintainable Rust transpilation. In the following sections, I describe each component of VERT in further detail.

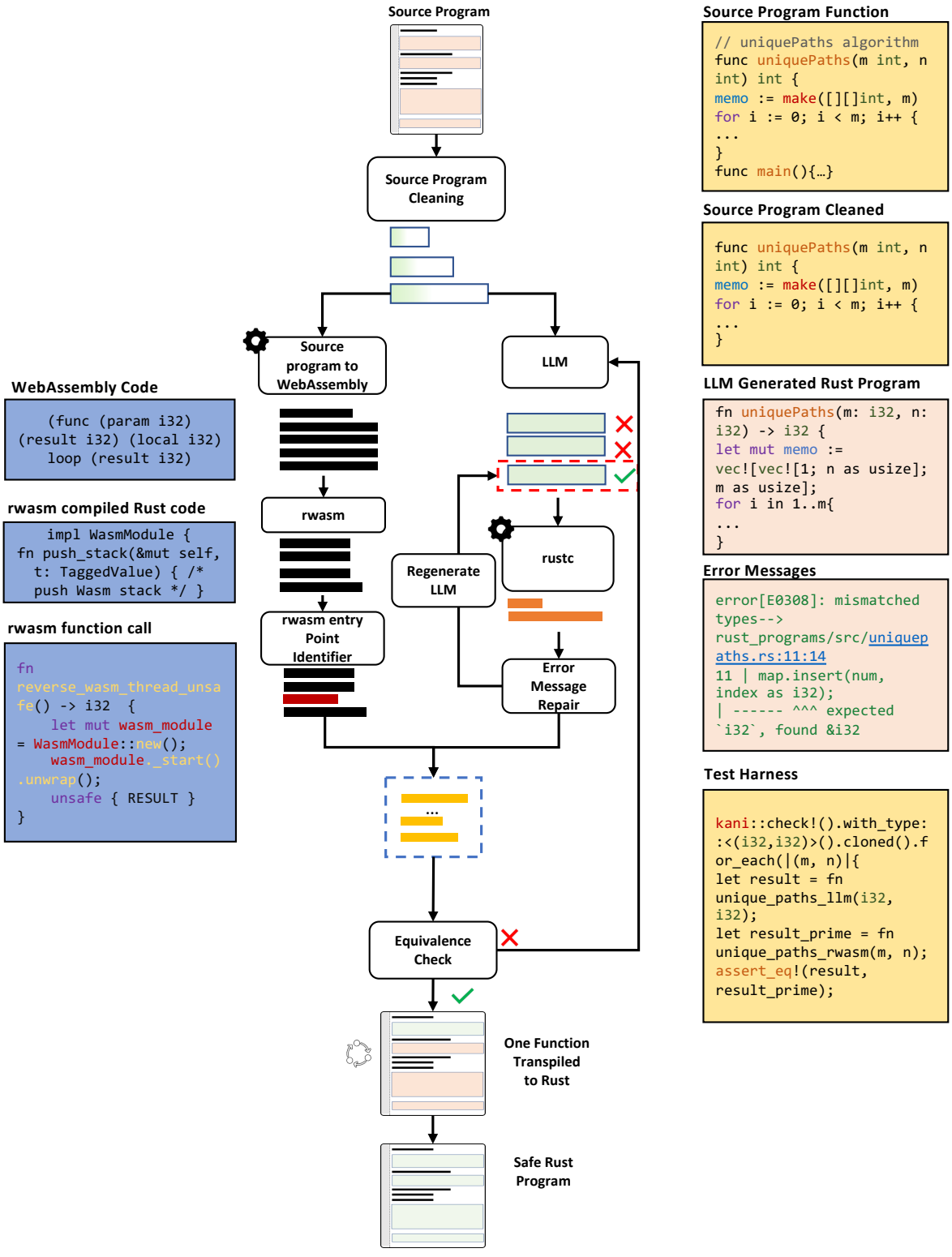

Figure 5.1: VERT's architecture, which takes as input a source program and produces a formally equivalent Rust program

```
error: mismatched closing delimiter: ']'
1  | fn roman_to_int(s: &str) -> i32 {
   |                                ^ unclosed delimiter
10 |     ]);
   |     ^ mismatched closing delimiter
```

Figure 5.2: Syntax error

```
error[E0308]: mismatched types-->
11 | map.insert(num,index as i32);| -- ^^^ expected '&i32', found i32
help: consider borrowing here: '&num'
```

Figure 5.3: Mismatched type error

5.1.1 Program repair on LLM output

LLMs often produce incorrect code. When prompting a LLM for Rust code, any slight mistake could cause the strict Rust compiler (rustc) to fail. Fortunately, rustc produces detailed error messages when compilation fails to guide the user to fix their program. I create an automatic repair system based on rustc error messages. For each error, I first classify the error into one of three main categories: syntax, typing, and domain specific.

As seen in Figure 5.2, an example syntax error generated by the LLM is the wrong closing delimiter. For rustc to successfully compile, all syntax errors must be resolved. I track the error code location (e.g., line 10 in Figure 5.2), and I use the rustc provided initial delimiter to guide my repair strategy. For this case, I know to use the right curly bracket \ to replace] on line 10.

Typing error messages in Rust generally have a similar structure. In particular, error messages are usually of the form expected type a , found b. Figure 5.3 shows the LLM generate a pass-by-reference variable &i32 while rustc expects a pass-by-value i32. Using the compiler message's error localization and suggestion line (characterized by the keyword help), I replace the variable num by &num.

Finally, domain-specific errors are compilation errors that are specific to the program. The error messages for domain-specific errors do not share the same structure, and therefore I only use the rustc error message suggestion line to generate a repair. In Figure 5.4, which shows the error message for an immutable assignment, the suggestion line indicates that if the variable x is converted to a mutable object, the immutable assignment error would be solved. Using this suggestion line, I replace x by mut x, and observe that the program compiles. It is often the case that even with the error message suggestion line, I cannot generate a repair that fixes all errors. In these cases, I regenerate a LLM output using the error as part of the new prompt and restart the process. my error-guided repair is significantly faster than the LLM generation (discussed in Section 5.3), so I only regenerate a LLM output after exhausting all rustc helper messages.

5.1.2 Transpilation Oracle Generation

Since the LLM output cannot be trusted on its own, I create an alternate trusted transpilation pipeline for generating a reference Rust program against which the LLM output is checked. The

```
error[E0384]: cannot assign to immutable argument 'x'
1  | fn reverse(x: i32) -> i32 {
   |- help: consider making this binding mutable: 'mut␣x'
...
16 |         x = x / 10;
   |         ^^^^^^^^^^ cannot assign to immutable argument
```

Figure 5.4: Immutable assignment error

```
func callReverse() int {
  result := reverse(123)
  if result == 321 {return 0}
  else {return 1}
}
```

Figure 5.5: An entry point for the `reverse` function

alternate pipeline does not need to produce maintainable code, but it needs to translate the source language into Rust using a reliably correct rule-based method. I use Wasm as the intermediate representation because many languages have compilers to Wasm, allowing it to serve as the common representation in the rule-based translation. Once the input programs are compiled to Wasm, I use rWasm [13], a tool that translates from Wasm to Rust by embedding the Wasm semantics in Rust source code. While the original authors intended rWasm as a sandboxing tool that leverages the memory safety properties of safe Rust, I use it to generate trusted Rust code with same semantics as the original input.

5.1.3 Mutation Guided Entry Point Identification

Given the assembly-like output of rWasm, I must perform analysis to identify the entry point of the rWasm transpiled function. VERT provides the option for the user to manually identify the entrypoint, but I can find it automatically using a simple heuristic, such as a function call or a single test case. I note that this heuristic could be generated automatically using LLMs or search-based software testing and thus I can assume an entrypoint generator in the source language. VERT uses a function call in the source language with constant inputs to the function to be transpiled and an assertion on the output of that function. One such function is given in Fig. 5.5.

I leverage this function call to identify the input and output of the function. While one option for such analysis is to perform decompilation, I find that a mutation-guided approach is sufficient for my purposes. In Fig. 5.5, I know that the input is 123 and the output is 321. Now, I wish to identify the equivalent constant in the rWasm output. While it is possible to just perform a linear scan of the rWasm output for this constant, that risks spurious matches, especially for simple types like i32. Instead, I guide this identification by leveraging the function call and mutating it. Suppose I swap 123 with 456 and 321 with 654 and re-transpile with rWasm. These constants will change, but the rest of the rWasm output remains the same. Taking the diff, I can identify inputs and outputs by what changed. The diff in the rWasm output is shown in Fig. 5.6.

```
fn func_4(&mut self, ) -> Option<i32> {
  let mut local_3 : i32 = 0i32;
  let mut local_4 : i32 = 0i32;
  v0 = TaggedVal::from(321i32);
  // mutant: v0 = TaggedVal::from(654i32);
  local_3 = v0.try_as_i32()?;
  v0 = TaggedVal::from(123i32);
  // mutant: v0 = TaggedVal::from(456i32);
  local_4 = v0.try_as_i32()?;
}
```

Figure 5.6: The difference between the original rWasm output and the mutated one (highlighted).

5.1.4 Equivalence Harness Generation

In my final step, I generate harnesses to check for equivalence given the input and output locations. I define equivalence here in functional terms: for all inputs, running both functions yields no crashes and identical outputs. To check this property holds, I automatically generate a wrapper to the Wasm function and a harness where the LLM-synthesized and wrapped rWasm functions are called with the same inputs, and the outputs are asserted to be equal. To ensure this equivalence holds for all inputs, I leverage property-based testing with random inputs and model-checking with symbolic inputs. For the remainder of this section, I refer to both of them together as "the input."

The wrapper consists of two parts: input injection, and output checking. I replace constants inputs with the inputs of the harness like input : i32. Instead of replacing the parameters to the function, I use globals in Rust to inject the inputs right at the location where constants used to be. An example is given in Fig. 5.7, with func_4 being the Wasm equivalent of the test. Note that, while this injection requires unsafe code, it is fine as this is only done in the oracle and the oracle is discarded once the equivalence is checked. I inject the baseline OUTPUT_1 and assert that the function returns 0. Since the function returns 0 when output equals the injected value, I know the functions returned the same value.

I note that while this approach is sound, it may falsely identify some equivalent programs as faulty due to semantic differences between Rust and the target language, or between the target language and rWasm embedding. I note two cases where I permit the analyst to add assumptions. First, when the input type is an unsigned integer, the analyst may assume nonzero values even though compilers may represent it as signed. Second, the analyst may restrict the range of strings to ASCII when the input is a C program to avoid crashing the Wasm with Rust's Unicode strings.

5.1.5 Equivalence Checking

With the equivalence checking harness built, I must now drive the harness and check that the equivalence property holds for all inputs. VERT provides two equivalence checking techniques with increasing levels of confidence and compute cost. I use existing tools that operate on Rust to prove equivalence between the LLM-generated and the rWasm-generated oracle Rust programs. I use Bolero, a Rust testing and verification framework that can check properties using both

```
static mut INPUT_1 = 0;
static mut OUTPUT_1 = 0;
impl WasmModule {
  /// returns 0 if the output matches
  fn func_4(&mut self, ) -> Option<i32> {
    // ...
    let mut local_3 : i32 = 0i32;
    let mut local_4 : i32 = 0i32;
    v0 = TaggedVal::from(unsafe {INPUT_1});
    local_3 = v0.try_as_i32()?;
    v0 = TaggedVal::from(unsafe {OUTPUT_1});
    local_4 = v0.try_as_i32()?;
    // ...
  }
}
/// equivalence-checking harness.
fn equvalence() {
  bolero::check!()
  .for_each(|(input: i32)| {
    let llm_fn_output = llm_generated_reverse();
    unsafe {
      INPUT_1 = input;
      OUTPUT_1 =  llm_fn_output;
    }
    let mut wasm_module = WasmModule::new();
    wasm_module._start().unwrap();
    assert!(wasm_module.func_4().unwrap() == 0);
  });
}
```

Figure 5.7: Equivalence-checking harness for `func_4`.

Property-Based Testing (PBT) [25] and Bounded Model Checking (BMC) [20, 21].

PBT

PBT works by randomly generating inputs to the function under test, running the harness with these inputs, and asserting the desired properties (e.g. equivalence between two functions). PBT repeatedly runs this procedure to check the property over the range of inputs. PBT is a valuable tool for catching implementation bugs, however it is generally infeasible to run PBT for long enough to exhaust all possible inputs to a program.

First, I run the equivalence-checking harness with PBT using Bolero up to the time limit, generating random inputs and checking equivalence of the outputs. If the candidate diverges from the oracle, then PBT will return the diverging input as a counterexample. If no counterexample is found within the time limit, I say this candidate passes PBT.

BMC

If the PBT stage succeeds, I now perform bounded verification with a combination of Bolero and Kani. I customize Bolero to perform model checking through Kani [81], a model-checker for Rust. When run with Kani, Bolero produces symbolic inputs rather than random concrete inputs. Executing the harness with symbolic inputs, I can cover the entire space of inputs in one run and the model-checker ensures the property holds for all possible inputs. Since symbolic execution does not know how many times loops are run, Kani symbolically executes loops up to an user-provided bound. To prove soundness of this bound, Kani uses *unwind checks* asserting that loop iteration beyond the bound is not reachable.

I run Kani with an unwinding bound of k (where $k = 10$) and *no unwind checks*. This means that paths up to k loop iterations is exhaustively explored, but any divergences between the candidate and the oracle with traces containing more than k loop iterations are missed. I run this phase for 120 seconds, and terminate with 3 potential results. First, Kani returns with a counterexample that causes the oracle and the candidate to diverge or one of the two to crash. Second, Kani does not return with an answer within the time limit, which I also consider to be a failure as we cannot establish bounded equivalence. Finally, Kani verifies that, limited to executions with at most k loop iteration, there are no divergences or crashes. I consider the third case alone to be successful.

Although for finite-input, deterministic programs, it is possible to perform verification with full exhaustive unwinding using Kani (i.e., fully verifying all k until exhaustion), I measure that the median time to exhaustively verify a Rust program within my selected dataset is 5 minutes, which is much longer than my established time limit of 120 seconds per program. I choose not to use full Kani verification as a metric due to time and compute constraints.

I support complex types through their primitive parts. Given a struct or enum, that Kani or Bolero does not initially support, I construct values of that type by abstracting the primitive parameters of that type and any required discriminants for enums. For types of finite size, this is sufficient. However, I provide bounded support for handling vector types. The challenge here is to vary the length of the vector in the rWasm output, which is done by having a fixed-length vector of varying inputs and then pruning the length down to the actual length dynamically. my

```
{Original code}
Safe Rust refactoring of above code in {language}.
Use the same function name, same argument and return types.
Make sure the output program can compile as a stand alone.
// If there exists counter examples from prior failed
// equivalence checking Test that outputs from inputs {counter_examples}
// are equivalent to source program.
```

Figure 5.8: LLM Prompt template.

approach is sound and complete for primitive types, and by extension, any type that comprises solely of primitive types such as tuples of primitives. For unbounded types like vectors, hashmaps and user-defined types containing such, VERT synthesizes harnesses that generate inputs up to the size encountered in the sample function call. As a limitation, any divergences that require larger vector than encountered will be missed.

5.1.6 Few-shot Learning

The main focus of this work is on verifying the output of LLMs for program transpilation, and not LLM prompt engineering. Therefore, I keep the prompts simple and short. Complicated and repeated querying of the same prompts do not provide additional benefits on the accuracy of outputs for small sized models, and too expensive for an average practitioner for industry sized models (i.e., Anthropic Claude). To achieve few-shot learning on my transpilation queries, each failed transpilation attempt provides its equivalence checking counter examples as a few-shot learning example for future transpilation attempts.

Figure. 5.8 shows my template for few shot learning. I start with querying the LLM to refactor the source code into safe Rust. Although I filter for safe Rust LLM output, I experimentally found that asking the LLM to always produce safe Rust gives more accurate results. I prompt the LLM to use the same argument and return types as the original, and can compile without external dependencies. Finally, I collect the counter examples from prior failed equivalence checks as part of the prompt. Specifically, I ask the LLM to consider the specific inputs that caused a test or verification failure from the previous iterations. I observed that providing specific inputs as information to the LLM results in subtle bug fixes within the program output.

5.2 Evaluation Setup

In this section, I present my evaluation setup for the following research questions.

RQ1. How does VERT perform vs. using the respective LLM by itself? I evaluate my technique's performance on a benchmark dataset, showing that VERT significantly increases the number of verified equivalent transpilations vs. using the LLM by itself.

RQ2. How does each component of VERT's approach impact its performance? I conduct an ablation analysis, which shows that my prompting and error-guided refinement helps produce more well-typed and more correct programs. I further measure the runtime performance of each part of VERT, showing that time costs of error-guided refinement is reasonable and VERT spends

most of the time in verification.

5.2.1 LLM Fine-tuning

The availability of Rust code in open source is scarce as compared code written in most other programming languages. Incoder [26] estimates that Rust is only the 20th most common language in their training database, which is a 159GB code corpus taken from Github BigQuery [1]. Due to the lack of Rust data available on open-source, I opt to not train a LLM targeted at Rust code generation. Instead, I directly use an off-the-shelf industry grade LLM, and also fine-tune on a separate open-source pretrained LLM. Specifically, I use Anthropic Claude-2 [2] for the industry grade LLM, and StarCoder [47] for the pretrained LLM.

I use light weight and parameter efficient adapter layers [33, 55, 93] for fine-tuning StarCoder. I collect 94 LeetCode type question solutions in C, C++, Go and Rust. Although there are existing code bases for all fmy languages, I find that LeetCode has the most consistent translation between other languages and Rust. I were able to collect 94 LeetCode questions of which have a direct translation between all 3 languages.

For each LeetCode type question, I have a corresponding source program (written in Go, C, or C++), and a target program (written in Rust). I encode all code words into tokens using the GPT-2 tokenizer. I fine-tune with 4 Transformer layers, 300 total epochs, and a final model dimension of 6144.

5.2.2 Benchmark selection

I draw my benchmarks from two sources. my first source is the benchmark set from TransCoder-IR [77], which is primarily made up of competitive program solutions. In total, this benchmark set contains 852 C++ programs, 698 C programs, and 343 Go programs. I choose this dataset to avoid potential data-leakage (i.e., LLM memorization) [10] in my evaluation. I note that the Rust programs produced by TransCoder-IR were released after June 2022, which is the training data cutoff date of my chosen LLMs [4, 47, 74]. I select programs from the TransCoder-IR dataset that can directly compile to Wasm using rWasm. After filtering, I collect a benchmark set of 569 C++ programs, 506 C programs, and 341 Go programs. These types of benchmarks are common for evaluating LLMs' coding ability. However, the programs themselves often do not make extensive use of pointers, so they do not adequately challenge VERT's ability to generate safe Rust.

To provide insight into VERT's ability to write safe rust, I gather 14 additional pointer-manipulating C programs from prior work on C to Rust transpilation [1, 24, 96]. I note, however, that the benchmarks in these prior works use open-source programs written before my chosen LLM's training data cutoff (June 2022). To avoid LLM data-leakage, I select and customize snippets from these C projects to transpile to Rust. I manually label the input output pairs for each snippet for verifying equivalence on the transpiled Rust programs. Many of the benchmarks I select involve multiple functions. The explicit goal when selecting benchmarks from these

[1] https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code
[2] https://www.anthropic.com/product

projects is to discover the limitations of VERT in terms of writing safe Rust, therefore I gather benchmarks of increasing complexity in terms of the number of pointer variables, and the number of functions in the benchmark. I present several complexity metrics for the benchmarks and discuss them in more detail in Section 5.3.

In total, I evaluate my approach on **569 C++** programs, **520 C** programs, and **341 Go** programs.

5.2.3 Evaluation Metrics

Neural machine translation (NMT) approaches use metrics that measure token similarity between the expected output and the actual output produced by the LLM. While these approaches are often meaningful when applied to natural language, for programming languages, small differences in the expected output and actual output could result in different compilation or run-time behavior. Conversely, two programs that share very few tokens (and hence have a very low text similarity score) could have identical compilation or run-time behavior. For programming languages, metrics based off of passing tests have been proposed. Roziere et al. [73] and Szafraniec et al. [77] use the computational accuracy (CA) metric, which counts a translation as correct if it passes a series of unit tests. However, there is no accepted standard for the number of required passing tests when using the CA metric. Furthermore, the CA metric does not take into account the quality or coverage of the unit tests.

To improve upon the metrics used for prior NMT approaches and remove the overhead of writing high-coverage unit tests, I use formal methods to measure the correctness of the output. In particular, I use property based testing (PBT) and bounded model checking (BMC). I insert the LLM-generated code and rWasm-generated code in an equivalence-checking harness that asserts equal inputs lead to equal outputs. An example of such a harness is given in Figure 5.7. Since the two metrics used are significantly slower than checking a series of unit tests, I set a time limit for my metrics. For both metrics, I set a 120 seconds limit. For PBT, no counterexamples within 120 seconds counts as success. For BMC, success requires establishing verified equivalence within 120 seconds. If either of the step fails, VERT terminates.

5.3 Results

I present results on the TransCoder-IR benchmarks in Table 5.1. I present VERT operating in three different modes. *Single-shot* means that VERT uses the LLM *once* to create a single candidate transpilation, and then proceeds directly to verification. If verification fails, then VERT does not attempt to regenerate. *few-shot* means that, if verification fails, then VERT will prompt the LLM to regenerate the transpilation repeatedly. In each iteration, I apply the syntactic repair described in Section 5.1.1 to the output of the LLM. Finally, *few-shot counter examples* means that I use counter examples produced by previous failed verification attempts as part of the LLM's few-shot learning, as described in Section 5.1.6. *few-shot counter examples* only works for instruction-tuned models. I re-prompt the LLM up to 20 times for few-shot modes. For each LLM and each mode of VERT, I report the number of transpilations that compiled and that passed the various verification modes. As seen in Table 5.1, I only perform *single-shot* for

Table 5.1: VERT performance across with different LLMs and modes.

LLM	Source Lang	Technique	Compiled	PBT	BMC
TranscoderIR(Baseline)	C++ (569)	Single-shot	107	23	3
	C (520)	Single-shot	101	14	1
	Go (341)	Single-shot	24	3	0
CodeLlama2 13B	C++ (569)	Few-shot	307	25	6
	C (520)	Few-shot	160	18	4
	Go (341)	Few-shot	104	15	2
StarCoder fine-tuned 15.5B	C++ (569)	Few-shot	253	79	8
	C (520)	Few-shot	179	76	4
	Go (341)	Few-shot	134	59	2
Claude-2 130B	C++ (569)	Single-shot	240	55	6
		Few-shot	539	292	41
		Few-shot counter examples (VERT)	**539**	**295**	**233**
	C (520)	Single-shot	239	49	6
		Few-shot	339	195	29
		Few-shot counter examples (VERT)	**339**	**209**	**193**
	Go (341)	Single-shot	126	26	3
		Few-shot	276	157	39
		Few-shot counter examples (VERT)	**317**	**195**	**159**

Transcoder-IR (baseline) to replicate results from prior work. I perform *few-shot* on CodeLlama2 and StarCoder fine-tuned to investigate the effectiveness of few-shot and rule-based repair on open-source, non-instruction tuned LLMs. Finally, I perform *single-shot, few-shot, and few-shot with counter examples* with Anthropic Claude-2 to investigate how each part of VERT impacts an instruction-tuned LLM's ability to perform Rust transpilation.

RQ1. How does VERT perform vs. using the respective LLM by itself?

As seen in table 5.1, VERT with Claude-2 compiles for 76% more programs for C++, 75% for C, and 82% for Go as compared to baseline (i.e., Transcoder-IR). VERT with Claude-2 can pass PBT for 49% more programs for C++, 37% for C, and 56% for Go as compared to baseline. VERT with Claude-2 can pass BMC for 40% more programs for C++, 37% for C, and 47% for

Table 5.2: VERT's average runtime per component for a Single-program translation

Component type	Component	Time (s)
LLM	Transcoder-IR	8
	CodeLlama-2	43
	Starcoder fine-tuned	45
	Anthropic Claude	30
Rust compilation	`rustc`	< 1
	Error guided	1
	`rwasm`	< 1
Testing and verification	PBT	25
	Bounded-ver.	52

Go as compared to baseline.

VERT with both CodeLlama2 and StarCoder fine-tuned also improve upon baseline on number of programs passing compilation, PBT, and BMC. I observe that few-shot learning with rule-based repair on general code-based LLMs can perform more accurate Rust transpilations than a LLM trained with transpilation as its main target.

RQ1 Summary

VERT with CodeLlama2, StarCoder fine-tuned, and Anthropic Claude-2 can produce more compiling, PBT, and BMC passing Rust transpilations than baseline. In particular, VERT with Claude-2 can pass BMC for 40% more programs for C++, 37% for C, and 47% for Go as compared to baseline.

RQ2. How does each component of VERT impact its performance? Table 5.1 shows the transpilation results across CodeLlama-2 and StarCoder fine-tuned in a few-shot setting. I observe that VERT with CodeLlama-2 and StarCoder fine-tuned improve over Transcoder slightly for compilable Rust translations. Since Rust is an underrepresented language in all LLMs trained on GitHub open-source repositories and The Stack dataset [38], I see that light-weight fine-tuning on a small dataset shows immediate improvement. In particular, I observe that StarCoder fine-tuned has fewer transpilations than CodeLlama-2 passing compilation, but more transpilations than CodeLlama-2 passing BMC. Fine-tuning with Rust code has an immediate impact on transpilation accuracy. StarCoder's results are limited by its ability to pass compilation, even with VERT's `rustc` error guided program repair in place. VERT with StarCoder fine-tuned compiles 47% fewer programs for C++, 41% fewer for C, and 63% fewer programs for Go as compared to VERT with Claude-2. While adding fine-tuning on Rust syntax increases the number of compilable translation generated, I observe that an industry-grade LLM with more trainable parameters and a larger training dataset performs significantly better for my metrics.

I observe that VERT using few-shot with either StarCoder fine-tuned or Claude-2 yields better transpilation across all my three languages and three metrics. In particular, few-shot with Claude-2 passes 43% more PBT checks for C++, 46% more for C, and 43% more for Go as compared to single-shot with Claude-2. Table 5.1 does not show single-shot results for CodeLlama-2 and

StarCoder fine-tuned as I observed no transpilations passing PBT. few-shot with Claude-2 passes 6% more BMC checks for C++, 4% more for C, and 12% more for Go as compared to single-shot with Claude-2. I find that the few-shot prompting for Claude-2 yields a greater improvement over single-shot compared to my repair technique. For C++ and C in particular, few-shot and repair with Claude-2 does not provide any additional passes on BMC as compared to only few-shot with Claude-2. I observe that few-shot learning with counter examples of failed previous verification attempts provides the largest improvements on BMC. Modern LLMs that are instruction-tuned can learn to generate more correct program when given specific test failures in few-shot settings.

Table 5.2 shows the average runtime of each of VERT's components across my entire evaluation dataset. I observe that in the non-timeout failure cases (i.e., Kani does not establish equivalence within 120s), Kani's BMC uses an average of 52 seconds per program, and Bolero's property testing uses an average of 25 seconds per program. Of the LLMs, both CodeLlama-2 and StarCoder use about 3 seconds per each prompt attempt, and Anthropic Claude-2 about 2 seconds. Not counting the failure cases (i.e., the LLM does not generate any program that can pass equivalence after 20 attempts), we observe an average of 15 tries before the LLM can achieve compilation. Transcoder-IR uses 8 seconds on average per transpilation, which I prompt only one time as the baseline of our evaluation.

RQ2 Summary

my ablation study shows that fine-tuning a LLM with Rust yields a higher accuracy of transpiled programs, as seen by a higher number of programs passing PBT and BMC by StarCoder fine-tuned compared to CodeLlama2. However, few-shot learning with counter examples provides the largest improvements on transpilation accuracy. Finally, I observe that VERT spends most of its runtime in verification.

5.4 Conclusion

Software evolution sometimes require entire project transpilation. As a case study, I investigate the potential for LLMs to translate between different programming languages, producing both compilable and test-passing programs. In this work, I study the combination of LLMs and formal verification to transpile verified and idiomatic Rust programs. I evaluate my tool *VERT* by transpiling 1,394 programs from C++, C, and Go. *VERT* with the Claude LLM can verify with bounded model checking for 40% more programs for C++, 37% for C, and 47% for Go as compared to baseline.

6 Proposed Work

As described in Chapter 3, Chapter 4, LLM can be a powerful tool for all stages of automated program repair: fault localization, patch generation, and plausible patch ranking. One opportunity from LLMAO is the capability of LLMs to perform security vulnerability detection without the usage of tests. Software security vulnerabilities allow attackers to perform malicious activities to disrupt software operations (i.e., security exploits). Since security exploits often occur during run-time, the independence of tests for LLM-based vulnerability detection shows promise. However, a key limitation of LLMAO is the size of potentially vulnerable programs is limited by the maximum size of LLM context windows. To circumvent the LLM maximum context window issue while expanding my target programs for vulnerability detection, my idea is to train LLMs using small snippets of code accompanied by important and relevant information of the code snippet, inferred by static analysis vulnerability explanations. To complete my dissertation, I aim to expand my prior work to detecting vulnerabilities from larger programs, and create tools that detect vulnerable functions across entire repositories instead of lines of code from single functions (i.e., LLMAO). I plan to incorporate the static analysis of a potentially vulnerable program into trainable objectives for LLM-based vulnerability detection.

Security vulnerabilities often span across multiple functions or files. Unlike logic defects, security exploits take advantage of weaknesses in the data flow across an entire project. Furthermore, vulnerabilities correspond to specific exploits or attacks on a system, and the nuances of a vulnerability only unfold when accompanied by vulnerability *explanations*. The nuances of security vulnerabilities have been a pain-point for prior static analysis and machine learning vulnerability detection tools. The specific properties of security vulnerabilities lead me to believe that the attention-based LLMs can leverage these nuances for greater detection effectiveness. Importantly, the presence of rich, detailed vulnerability explanations available online leads me to believe a LLM can gain an understanding of how and why vulnerability can exist, and how that could lead to a security exploit. For my proposed work on security vulnerabilities, I plan to fine-tune a LLM capable of recognizing vulnerability explanations, and finally achieve state-of-the-art vulnerability detection on across entire functions and files.

I propose to build LLMs that train on multiple dimensions of vulnerability information in combination with dataflow-inspired graph neural networks (GNNs). Multitask learning enables a model to learn shared knowledge and patterns simultaneously, typically leading to improved generalization and accuracy. My proposed approach is based on both recent advances in LLM research that enable fine-tuning on relatively small datasets, and the insights that (1) joint fine-tuning encompassing both code and vulnerability explanations can enhance performance compared to solitary code fine-tuning methods, and (2) most security vulnerabilities entail specific

and often subtle information flow, but training language models on either code or explanations alone will not capture key relations between values and data propagated through a potentially vulnerable program. Representing the program as a graph is therefore essential, in conjunction with the multi-task learning. I plan to extract a subset of previously established security vulnerability datasets BigVul and PreciseBugs for model evaluation, and performing ablation studies on different combinations of pre-trained LLMs and security vulnerability explanations.

The BigVul and PreciseBugs datasets are focused on the C programming language, and do not include vulnerabilities spanning across repositories. Both datasets include a single function as a single data entry, and a label indicating whether the function is involved in a security vulnerability. To ensure generalization of my method for detecting vulnerabilities across many functions and files, I plan to further expand my LLM-based vulnerability detector on JavaScript. In particular, I plan to build models focusing on detecting exploits in the Node.js ecosystem. Each Node.js package serves as a building block for developers to create applications, and has a set of public APIs: entry point functions that can be called from other packages. As Node.js becomes more popular, the ecosystem has become an attractive target of attackers. Due to these entry points across files (called sinks in Node.js), Node.js is an obvious candidate to further expand on my proposed vulnerability detector. Dynamic taint analyses aim to find a flow of information from attacker-controlled inputs to a package's entry point to sensitive APIs (i.e., sinks). As the final proposed work of my thesis, I propose to incorporate dynamic taint analysis with manually confirmed security exploits as GNNs, and compare my GNN+LLM technique to state-of-the-art taint-analysis exploit detection techniques.

7 Proposed Timeline

I propose the following schedule with an expected defense date of May 2025. At the time of writing this thesis proposal I have completed and published the LLM-based fault localization and APR tools described in Chapter 3 and Chapter 4. I have implemented and completed all evaluation for the transpilation tool VERT described by Chapter 5, and submitted to ICSE 2025. I have made initial investigations into the work proposed in Section 6.

- **September-October 2024**
  - Complete thesis proposal.
  - Literature review for security vulnerability detection from using traditional ML and recent LLMs.
  - Complete data collection for novel vulnerability data.
- **October-December 2024**
  - Set up vulnerability explanation model for fine-tuning.
  - Train model for security vulnerability detection.
  - Evaluate prior work on vulnerability detection.
- **December 2024-March 2025**
  - Complete vulnerability detection research.
  - Begin writing final thesis document.
- **March-June 2025**
  - Finish works in progress, and resubmit possibly rejected papers.
  - Finish writing thesis document.
  - Prepare for defense.
- **June 2025**
  - Defend and graduate.

8 Conclusion

In summary, this thesis proposal describes three LLM based tools for software fault localization, patch efficiency, and automatic programming language transpilation. My work is the first to use LLMs for all stages of program repair, as well as the first to propose tooling on combining LLMs with automated verification on the task of program transpilation.

The implementations of the techniques completed in this thesis proposal are open to the publicly and already used by researchers studying related work. For the remainder of my PhD, I propose to expand my fault localization work to detect entire functions with as potentially vulnerable across multiple files in a larger repository.

Bibliography

[1] C2rust. https://c2rust.com/. 2.1, 5.2.2

[2] bindgen. https://github.com/rust-lang/rust-bindgen. 2.1

[3] citrus. https://gitlab.com/citrus-rs/citrus. 2.1

[4] claude. https://www.anthropic.com/index/introducing-claude. 5.2.2

[5] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46. IEEE, 2006. 2.1, 3.2.2

[6] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007. 1, 2.1

[7] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. An algebra of alignment for relational verification. *Proceedings of the ACM on Programming Languages*, 7(POPL):20:573–20:603, January 2023. doi: 10.1145/3571213. URL https://dl.acm.org/doi/10.1145/3571213. 2.1

[8] Simone Balloccu, Patrícia Schmidtová, Mateusz Lango, and Ondřej Dušek. Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms. *arXiv preprint arXiv:2402.03927*, 2024. 4

[9] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994. 3.1.2

[10] Stella Biderman, USVSN PRASHANTH, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. Emergent and predictable memorization in large language models. *Advances in Neural Information Processing Systems*, 36, 2024. 5.2.2

[11] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022. 1, 2.1

[12] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022. 3, 3.1.1

[13] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. *Provably-Safe* multilingual software sand-

boxing using *WebAssembly*. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1975–1992, 2022. 2.1, 5.1.2

[14] Peter F Brown, Vincent J Della Pietra, Peter V Desouza, Jennifer C Lai, and Robert L Mercer. Class-based n-gram models of natural language. *Computational linguistics*, 18(4): 467–480, 1992. 4

[15] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023. 1

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1, 2.1

[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1, 3, 3.1.1, 3.1.1

[18] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound Loop Superoptimization for Google Native Client. *ACM SIGARCH Computer Architecture News*, 45(1):313–326, April 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037754. URL `https://dl.acm.org/doi/10.1145/3093337.3037754`. 2.1

[19] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1027–1040. Association for Computing Machinery, June 2019. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314596. URL `https://doi.org/10.1145/3314221.3314596`. 2.1

[20] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 2001. 5.1.5

[21] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. doi: 10.1007/978-3-540-24730-2_15. URL `https://doi.org/10.1007/978-3-540-24730-2_15`. 5.1.5

[22] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356, 2016. 2.1

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 2.1

[24] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021. 2.1,

5.2.2

[25] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997. 5.1.5

[26] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022. 5.2.1

[27] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2022. URL `https://arxiv.org/abs/2204.05999`. 2.1, 4.2.1

[28] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020. 1

[29] Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 654–665, 2022. 2.1, 4, 4.2.2, 4.2.4

[30] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. 4

[31] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020. 2.1

[32] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022. 3.2.6

[33] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*, 2023. 5.2.1

[34] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*, 2023. 1

[35] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014. 3, 3.2.1, 4.2.2

[36] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. 1

[37] Misoo Kim, Youngkyoung Kim, Kicheol Kim, and Eunseok Lee. Multi-objective optimization-based bug-fixing template mining for automated program repair. In *Proceed-*

ings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pages 1–5, 2022. 4

[38] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022. 5.3

[39] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176, 2016. 3.2.4

[40] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*, 2022. URL `https://openreview.net/forum?id=rHlzJh_b1-5`. 2.1

[41] Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet. Self-supervised learning to prove equivalence between straight-line programs via rewrite rules. *IEEE Transactions on Software Engineering*, 49(7):3771–3792, July 2023. ISSN 1939-3520. doi: 10.1109/TSE.2023.3271065. 2.1

[42] Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. Automated program repair using generative models for code infilling. In *International Conference on Artificial Intelligence in Education*, pages 798–803. Springer, 2023. 1

[43] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014. 2.1

[44] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011. 2.1

[45] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019. 1, 4

[46] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. 4.2.1

[47] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. 5.2.1, 5.2.2

[48] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019. 2.1

[49] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019. 1, 3.2.2, 3.2.3, 3.2.4, 3.2.7

[50] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection

via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360588. URL https://doi.org/10.1145/3360588. 3.2.1

[51] Yi Li, Shaohua Wang, and Tien Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pages 661–673. IEEE, 2021. 3.2.2, 3.2.3

[52] Yi Li, Shaohua Wang, and Tien N Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 511–523, 2022. 2.1, 4.1.2

[53] Yi Li, Shaohua Wang, and Tien N Nguyen. Fault localization to detect co-change fixing locations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 659–671, 2022. 1, 3.2.4

[54] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. In Rust we trust: a transpiler from unsafe C to safer Rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 354–355, 2022. 2.1

[55] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022. 5.2.1

[56] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42, 2019. 1, 4, 4.1.2

[57] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 615–627, 2020. 4, 4.2.3, 4.2.4

[58] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016. 1

[59] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021. 2.1

[60] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021.

1, 3.2.1, 3.2.3, 3.2.4, 3.2.7

[61] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701. ACM, 2016. 4

[62] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1169–1180, 2022. 2.1, 4.1.2

[63] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1169–1180, 2022. 3.2.1, 3.2.2, 3.2.3

[64] Marcus J Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. Beyond accuracy: Evaluating self-consistency of code llms. In *The Twelfth International Conference on Learning Representations*, 2023. 1

[65] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022. 3, 3.1.1

[66] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. 1

[67] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011. 3.2.4

[68] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. Does genetic programming work well on automated program repair? In *2013 International Conference on Computational and Information Sciences*, pages 1875–1878. IEEE, 2013. 2.1

[69] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015. 2.1

[70] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016. 2.1, 4

[71] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016. 3.2.1

[72] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming*

language design and implementation, pages 419–428, 2014. 2.1

[73] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33, 2020. 1, 5.2.3

[74] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. 5.2.2

[75] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1): 61–80, 2008. 2.1

[76] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 532–543, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786825. URL https://doi.org/10.1145/2786805.2786825. 2.1, 4

[77] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*, 2022. 5.2.2, 5.2.3

[78] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–34, 2023. 2.1, 4, 4.2.2, 4.2.4

[79] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. 2.1, 4.2.1

[80] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. *Natural language processing with transformers*. O'Reilly Media, Inc., 2022. 3, 3.1.1

[81] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 321–330, 2022. 5.1.5

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 3.1.1, 3.1.2

[83] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013. 2.1

[84] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Con-

stance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1556–1560, 2020. 3.2.1, 3.2.7

[85] Robert F Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007. 3.2.7, 3.2.7

[86] Dongrui Wu and Jerry M Mendel. Patch learning. *IEEE Transactions on Fuzzy Systems*, 28(9):1996–2008, 2019. 1

[87] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023. 1

[88] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023. 2.1, 4, 4.1.2

[89] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*, pages 789–799, 2018. 1, 2.1, 4, 4.2.4

[90] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*, 2024. 1

[91] Aidan ZH Yang, Ruben Martins, Claire Le Goues, and Vincent J Hellendoorn. Large language models for test-free fault localization. *arXiv preprint arXiv:2310.01726*, 2023. 1

[92] Aidan ZH Yang, Sophia Kolak, Vincent J Hellendoorn, Ruben Martins, and Claire Le Goues. Revisiting unnaturalness for automated program repair in the era of large language models. *arXiv preprint arXiv:2404.15236*, 2024. 1

[93] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024. 5.2.1

[94] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. A large-scale empirical review of patch correctness checking approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1203–1215, 2023. 2.1, 4

[95] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 48(8):2920–2938, 2021. 2.1

[96] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. Ownership guided C to Rust translation. In *Computer Aided Verification (CAV)*, volume 13966 of *LNCS*, pages 459–482. Springer, 2023. 2.1, 5.2.2

[97] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective

vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019. 3.2.1, 3.2.7