Integrating Program Analysis with Language Models for Software Evolution

Aidan Z.H. Yang

August, 2025

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

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, we propose mechanisms to improve the utility of LLMs for software evolution by using and combining LLMs with prior APR and program verification techniques. Specifically, we build LLM-based software engineering tools for fault localization, program repair, and program transpilation.

My thesis statement is: Program analysis complements large language models (LLMs) for software maintenance by providing structured software signals that include bidirectional code context, formal verification of code properties, and dataflow properties. These signals can be integrated with LLMs beyond autoregressive generation. Explicitly integrating analysis-derived signals with LLMs consistently outperforms either pure learning-based or pure analysis-based approaches across key software engineering tasks in real-world repositories.

To support this statement, we make the following contributions.

I first propose and evaluate a bidirectional fine-tuning technique that enables a previously left-to-right LLM to locate and rank faulty lines of code. We built the LLM-based fault localization technique without depending on preciously written tests, and so the tool can also detect runtime 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, we used LLM entropy values to improve three stages of program repair: fault localization, patch testing efficiency, and plausible patch ranking. Finally, we created a tool for fully automated Rust function transpilation using LLMs and verification harnesses. To support our claims, we evaluated all our LLM-based software evolution tools on real world bugs, security vulnerabilities, and target transpilation repositories.

I extend our study of LLM-based fault localization by adding mechanisms for multi-task LLM instruction-tuning, and evaluating the model on real-world security vulnerabilities in large repositories. Specifically, the completed fault localizer only detects faults on a line level, and does not take into account various nuances of security vulnerability explanations. 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. The completed vulnerability detector attempts to detect vulnerabilities that span across different files across aa larger repository.

Acknowledgments

I would like to thank my PhD advisors Claire Le Goues, Ruben Martins, and Vincent Hellendoorn for their mentoring, guidance, and patience. My parents Yin Chen and Yong Yang for encouraging me to start this PhD journey from an early age. My collaborators and mentors: Ansong Ni, Brandon Paulsen, Daniel Kroening, Daniel Ramos, Darion Cassel, He Ye, Haoye Tian, Joey Dodds, Limin Jia, Ronghao Ni, Soongho Kong, Sophia Kolak, Yoshi Takashima, and Zhizhen Qin. My partner Samantha Hong, and everyone at S3D and SquaresLab for supporting me through my PhD.

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) [60, 76, 79, 133, 136]. 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) [2, 65, 70, 81] 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 [8, 15]. DL therefore affords promising opportunities for program repair [46, 57, 79, 133, 134] and fault localization [139]. The most effective DL models for both natural language and code related tasks are large language models (LLMs), such as Codex [16] and GPT-4 [12]. 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 [16] 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 [49], 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), we 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.

Judgment 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. We 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 [143]. 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 [106]. However, directly prompting a LLM comes with no formal guarantees that the resulting code will maintain input-output equivalence with the original [88, 98, 137]. Due to the unique nature of code transpilation, in which program equivalence can be represented by equivalence input-output pairings, we 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 while LLMs excel at autoregressive code generation, this generation paradigm is not directly applicable to many software maintenance tasks that require discriminative reasoning, correctness guarantees, or understanding of program semantics. Software maintenance tasks benefit from integrating program analysis techniques that provide structured signals—such as bidirectional context, formal verification, and dataflow properties—with LLM representations beyond simple token generation.

1.1 Thesis Statement

Program analysis complements large language models (LLMs) for software maintenance by providing structured software signals that include bidirectional code context, formal verification of code properties, and dataflow properties. These signals can be integrated with LLMs beyond autoregressive generation. Explicitly integrating analysis-derived signals with LLMs consistently outperforms either pure learning-based or pure analysis-based approaches across key software engineering tasks in real-world repositories.

1.2 Contributions

This dissertation presents a comprehensive investigation of integrating Large Language Models with program analysis techniques for software evolution. Through four major contributions, I demonstrate that hybrid neural-symbolic approaches consistently outperform pure neural or symbolic methods across multiple software maintenance tasks. Each contribution was novel at the time of publication and has been empirically validated on real-world software systems.

1.2.1 Bidirectional Fine-Tuning for Fault Localization

I introduce LLMAO (Large Language Models for Fault Localization), a novel approach that enables test-free fault localization through bidirectional adapter fine-tuning [139]. While state-of-the-art LLMs are trained in a left-to-right manner for code generation, this training paradigm limits their effectiveness for discriminative tasks like fault localization, where understanding the full context of code is essential. LLMAO addresses this limitation by training lightweight bidirectional adapters on top of frozen left-to-right language models, adding relatively few parameters while enabling the model to leverage bidirectional context for identifying faulty code.

The key novelty of this work lies in demonstrating that pretrained left-to-right language models already contain rich representations about code suspiciousness, which can be effectively leveraged through bidirectional adapters trained on small datasets of real bugs. Unlike traditional fault localization techniques that rely on test execution and coverage information, LLMAO operates without any test cases, making it applicable to scenarios where tests are unavailable or of poor quality. This test-free approach also enables LLMAO to detect runtime security vulnerabilities that may not be covered by existing test suites.

I evaluate LLMAO on three benchmark datasets: 395 bugs from Defects4J (Java), 493 bugs from BugsInPy (Python), and 5,260 security vulnerabilities from Devign (C). The results demonstrate that LLMAO outperforms existing state-of-the-art deep learning-based fault localization techniques including DeepFL, GRACE, and TransferFL across all benchmarks. The approach requires significantly less data preprocessing overhead compared to prior work and can be easily adapted to different programming languages by replacing the underlying pretrained model.

Artifact: The implementation and datasets are available at `https://github.com/squaresLab/LLMAO`.

1.2.2 Entropy-Based Uncertainty Quantification for Automated Program Repair

I demonstrate that LLM entropy values—a measure of model uncertainty during generation—can be leveraged to improve all three critical stages of automated program repair: fault localization, patch testing efficiency, and plausible patch ranking [143]. This work revisits the concept of code naturalness in the era of large language models, showing that while LLMs perceive buggy code as unnatural (high entropy) and tend to generate natural (low entropy) correct code, this property can be systematically exploited beyond simple patch generation.

The novelty of this contribution lies in introducing the concept of "entropy-delta", the change in code naturalness when a patch is applied, and demonstrating its utility across the entire APR pipeline. For patch generation efficiency, I show that using entropy to prioritize candidate patches for testing reduces the average number of patches that must be evaluated by 24 patches per bug. For patch correctness assessment, entropy-based ranking achieves 49% better Top-1 accuracy compared to the state-of-the-art patch ranker Shibboleth, without requiring any project-specific training data.

Importantly, this work establishes that combining LLM capabilities with traditional APR techniques produces superior results compared to using either approach in isolation. The entropy-based approach is agnostic to the patch generation method (template-based, symbolic, or ML-

based) and can be applied to patches produced by any APR tool. I evaluate this approach on the Defects4J benchmark, demonstrating consistent improvements across multiple APR stages and showing that the technique remains effective for patches produced by modern ML-based repair tools.

Artifact: The implementation and experimental data are available at `https://github.com/squaresLab/entropy-apr-replication`.

1.2.3 Verified Equivalent Rust Transpilation with LLMs

I present VERT (Verified Equivalent Rust Transpilation), a novel framework that combines LLMs with formal verification to produce functionally correct and idiomatic Rust code from source programs in other languages [141]. While LLM-based transpilation approaches typically produce more natural and maintainable code than rule-based transpilers, they provide no guarantees about correctness. Conversely, rule-based transpilers like C2Rust can theoretically produce correct transpilations but often generate unidiomatic code that relies heavily on Rust's unsafe subset.

VERT bridges this gap by using LLMs for initial code generation while employing formal verification to ensure correctness. The key innovation is a dual-path verification approach: VERT generates both an LLM-produced Rust transpilation and a reference implementation compiled through WebAssembly (Wasm), then uses equivalence checking to verify that both implementations produce identical outputs. This approach requires only that a WebAssembly compiler exists for the source language, making it applicable to most major programming languages.

The framework includes an automatic repair system that uses Rust compiler error messages to iteratively fix common compilation errors in LLM-generated code, including syntax errors, type mismatches, and domain-specific issues. This repair mechanism significantly reduces the need for LLM regeneration while maintaining the naturalness of the generated code.

I evaluate VERT on real-world Go repositories, demonstrating that it achieves 100% functional correctness through formal verification while producing code that is more idiomatic and maintainable than code generated by C2Rust. The evaluation shows that VERT successfully transpiles complex real-world functions while maintaining both correctness guarantees and code quality.

Artifact: The VERT implementation and evaluation benchmarks are available at `https://github.com/YoshikiTakashima/vert`.

1.2.4 Multi-Task Security Vulnerability Detection

I extend the fault localization work to security vulnerability detection through multi-task instruction-tuning of LLMs, addressing key limitations of line-level fault localization for security vulnerabilities that often span multiple files [142]. This work introduces MSIVD (Multi-task Self-Instruct Vulnerability Detection), which trains LLMs to simultaneously identify vulnerable code and explain exploitation vectors, enabling more comprehensive vulnerability detection across large repositories.

The novelty of this contribution lies in three key innovations: (1) a self-instruct dialogue-based data augmentation approach that transforms classification-based vulnerability datasets into multi-round conversations between teacher and student, enabling chain-of-thought reasoning; (2)

multi-task learning that jointly optimizes for vulnerability detection and explanation generation, allowing the model to learn shared knowledge and patterns that improve generalization; and (3) integration with graph neural networks (GNNs) based on control flow information to capture non-local and subtle information flow patterns essential for security vulnerability detection.

To address concerns about data contamination in LLM evaluation, I curate a novel vulnerability dataset with samples exclusively filtered to be vulnerabilities identified after the pre-trained model's training cutoff date. This ensures that evaluation results reflect genuine model capabilities rather than memorization of training data.

I evaluate MSIVD on established vulnerability datasets (BigVul and PreciseBugs) as well as the novel post-training-cutoff dataset, demonstrating state-of-the-art performance. The multi-task learning approach shows significant improvements over single-task vulnerability detection, and the integration with GNNs enables effective detection of vulnerabilities that span across multiple files and functions.

Artifact: The MSIVD implementation, datasets, and trained models are available at `https://github.com/squaresLab/multitask-vuln-paper`.

2 Background and Related Works

This chapter provides foundational background on the key technologies that underpin this dissertation: Large Language Models (LLMs) and their application to code-related tasks, and program analysis techniques. Chapter-specific related work and detailed technical background are presented in their respective chapters.

2.1 Large Language Models

Large Language Models (LLMs) are neural networks trained on vast amounts of text data to predict and generate natural language. In recent years, LLMs have been increasingly applied to code-related tasks, leveraging the observation that source code shares many properties with natural language while also exhibiting unique structural characteristics.

2.1.1 Transformer Architecture

Modern LLMs are built on the Transformer architecture [125], which uses self-attention mechanisms to process sequential data. Unlike earlier recurrent neural networks that process tokens sequentially, Transformers can attend to all positions in a sequence simultaneously, enabling more effective learning of long-range dependencies.

The core innovation of the Transformer is the attention mechanism, which computes weighted representations of input tokens based on their relationships to other tokens in the sequence. For a sequence of tokens, the model learns to assign attention weights that determine how much each token should influence the representation of every other token. This mechanism allows the model to capture complex dependencies and contextual relationships in the input.

Most LLMs for code are trained in a left-to-right (causal) manner, where each token is predicted based only on preceding tokens. This training objective, while effective for code generation tasks, creates representations where each token's embedding is conditioned only on its left context. This limitation motivates some of the technical contributions in this dissertation, particularly for discriminative tasks like fault localization that benefit from bidirectional context.

2.1.2 LLMs for Code

Language models have been successfully applied to various code-related tasks including code completion [25, 34], code generation from natural language descriptions [105], and program repair. Large Language Models such as Codex [15], GPT-Neo [8], and Llama-2 [118] have

significantly improved performance on these tasks by scaling up both model parameters and training data.

These models are typically trained on large corpora of source code collected from open-source repositories. The training process teaches the model to predict the next token in a code sequence, which implicitly captures patterns of correct, idiomatic code. As a result, well-trained LLMs tend to assign higher probability (lower entropy) to correct code and lower probability (higher entropy) to buggy or unusual code—a property known as code naturalness [41, 103].

The scale of modern LLMs is a key factor in their effectiveness. Models with billions of parameters trained on hundreds of billions of tokens can capture subtle patterns in code structure, naming conventions, and programming idioms. This capability makes them powerful tools for software engineering tasks, though their effectiveness varies depending on whether the task aligns with their training objective of left-to-right generation.

2.2 Program Analysis

Program analysis encompasses a broad range of techniques for automatically reasoning about program behavior and properties. These techniques can be broadly categorized as static analysis (analyzing code without executing it) and dynamic analysis (analyzing program behavior during execution).

2.2.1 Static Analysis

Static analysis examines source code or intermediate representations to infer properties about program behavior without executing the program. Common static analysis techniques include control flow analysis, data flow analysis, and abstract interpretation. These techniques can identify potential bugs, security vulnerabilities, and violations of coding standards.

For the work in this dissertation, static analysis techniques are particularly relevant for understanding program structure and data dependencies. Control flow graphs (CFGs) represent the possible execution paths through a program, while data flow analysis tracks how values propagate through variables and operations. These representations provide structured information about program semantics that complements the pattern-based understanding captured by LLMs.

2.2.2 Dynamic Analysis

Dynamic analysis observes program behavior during execution to gather information about actual program states and behaviors. This includes techniques like dynamic taint analysis, which tracks how data flows through a program during execution, and test-based analysis, which uses test execution results to infer program properties.

Dynamic taint analysis is particularly important for security vulnerability detection, as it can track how untrusted input propagates through a program to potentially dangerous operations. This technique labels certain data as “tainted” (e.g., user input) and tracks how this taint spreads through program operations, flagging potential vulnerabilities when tainted data reaches sensitive operations without proper sanitization.

2.2.3 Formal Verification

Formal verification uses mathematical techniques to prove that a program satisfies certain properties or specifications. Unlike testing, which can only demonstrate the presence of bugs in tested scenarios, formal verification can provide guarantees about program correctness for all possible inputs (within the scope of the verification).

Techniques like bounded model checking [19, 20] and property-based testing [32] can verify specific properties of programs, such as the absence of certain classes of errors or the equivalence of two program implementations. Bounded model checking explores all possible program states up to a specified depth bound, systematically checking whether any execution path violates a given property. While this approach does not provide full formal guarantees for unbounded executions, it offers significantly stronger assurances than traditional testing by exhaustively exploring the bounded state space. Property-based testing, on the other hand, automatically generates test cases based on specified properties or invariants that the program should satisfy, testing these properties across a wide range of inputs. Though property-based testing relies on sampling rather than exhaustive verification, it provides more comprehensive coverage than manually written test cases and can uncover edge cases that developers might not anticipate. Both techniques occupy a valuable middle ground between lightweight testing and heavyweight formal verification, offering practical approaches to increase confidence in program correctness without the complexity and computational cost of full formal proofs. These techniques are particularly valuable when correctness guarantees are essential, such as in the transpilation work presented in this dissertation.

2.3 Integration of LLMs and Program Analysis

A central theme of this dissertation is that LLMs and program analysis techniques have complementary strengths and limitations. LLMs excel at capturing patterns and generating natural, idiomatic code but lack formal understanding of program semantics and correctness. Program analysis provides rigorous reasoning about program properties but may struggle with the flexibility and naturalness that LLMs offer.

By integrating these approaches—using LLMs for their pattern recognition and generation capabilities while employing program analysis for semantic understanding and correctness verification—we can create tools that outperform either approach in isolation. The following chapters demonstrate this integration across multiple software evolution tasks: fault localization (Chapter 3), automated program repair (Chapter 4), program transpilation (Chapter 5), and security vulnerability detection (Chapters 6 and 7).

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.

This chapter demonstrates how program analysis complements LLMs for fault localization by providing bidirectional code context as a structured signal. While autoregressive LLMs excel at code generation through left-to-right prediction, this paradigm is not directly applicable to discriminative tasks like fault localization that require understanding code from both prefix and suffix context. This chapter shows that integrating bidirectional context—a form of program analysis—with LLM representations enables effective fault localization beyond autoregressive generation, supporting the thesis that analysis-derived signals complement LLMs for software maintenance tasks.

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 [9, 16, 95, 119] 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.

We 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* [47], without updating the underlying LLM. We 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 our adapters to find bugs while requiring just a few hundred training samples for pretraining. our approach yields better fault localization performance than prior work while requiring significantly less data preprocessing overhead. Importantly, our approach does not use test cases at all, and therefore does not depend on test code quality for its performance.

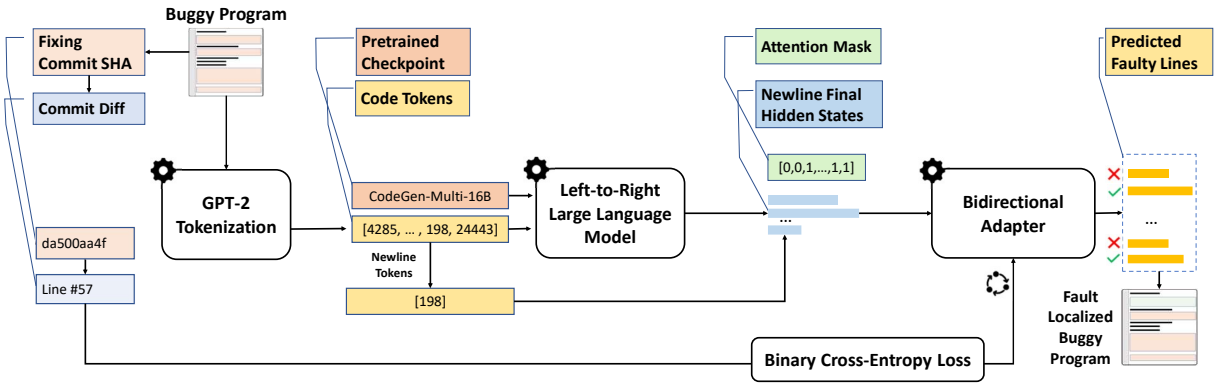

Figure 3.1: *LLMAO*'s architecture, which takes as input a buggy program and produces a list of suspiciousness scores for each code line

3.1 Background and Related Work

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) [1, 2] 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) [109] for fault localization. GRACE [80], DeepFL [64], and DEAR [69] encode the code AST and test coverage as graph representations before training deep learning models for fault localization. TransferFL [86] 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.

3.2 LLMAO: Large Language Models for Fault Localization

In this section, we discuss the key ideas behind our 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, we first tokenize the input and then provide it to a pretrained left-to-right LLM. From this LLM, we obtain one (high-dimensional) *vector representation* per line, which I provide to a small bidirectional model that predicts bugginess probabilities for each line. We 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 our language modeling procedure, which I describe in detail in Section 3.2.2. In the following sections, we describe each component of *LLMAO*.

3.2.1 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 [125]. 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 our bidirectional language model. Auto-regressive and left-to-right LMs [9, 16, 95, 119] 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.

our 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 [95] 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. [36] 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 [16] 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, we 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. We pair these with their corresponding location (i.e., line #5 of a 50 line file) and save these to disk.

To train our model, we load these encoded lines in batches, where I retrieve samples of up to 128 contiguous newline states at a time. We 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, we sample a series of 128 line windows that cover each faulty line in the file. Specifically, we 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. We then mark all faulty lines in this segment as covered and repeat until all lines are covered by at least one segment. We 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 our model to memorize certain index locations as faulty lines. This enables our technique to handle inputs longer than 2048 tokens.

3.2.2 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 bottle-necking phenomenon [6] 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 [125].

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 representations of later and earlier lines, thereby generating a new, bidirectionally aware representation for each line of code. We can do so by removing the causal attention mask that normally prevents the exchange of information with "future" tokens in our added layers. In our case, we 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 [125]. Concretely, our approach involves five steps, visualized in Figure 3.2:

Step 1: I start with a series of code tokens $C = [c_0, c_1, \ldots, c_N]$. We 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 $M \ll N$. We conjecture that these tokens' states reasonably accurately capture the information of their line in the file's prefix context.

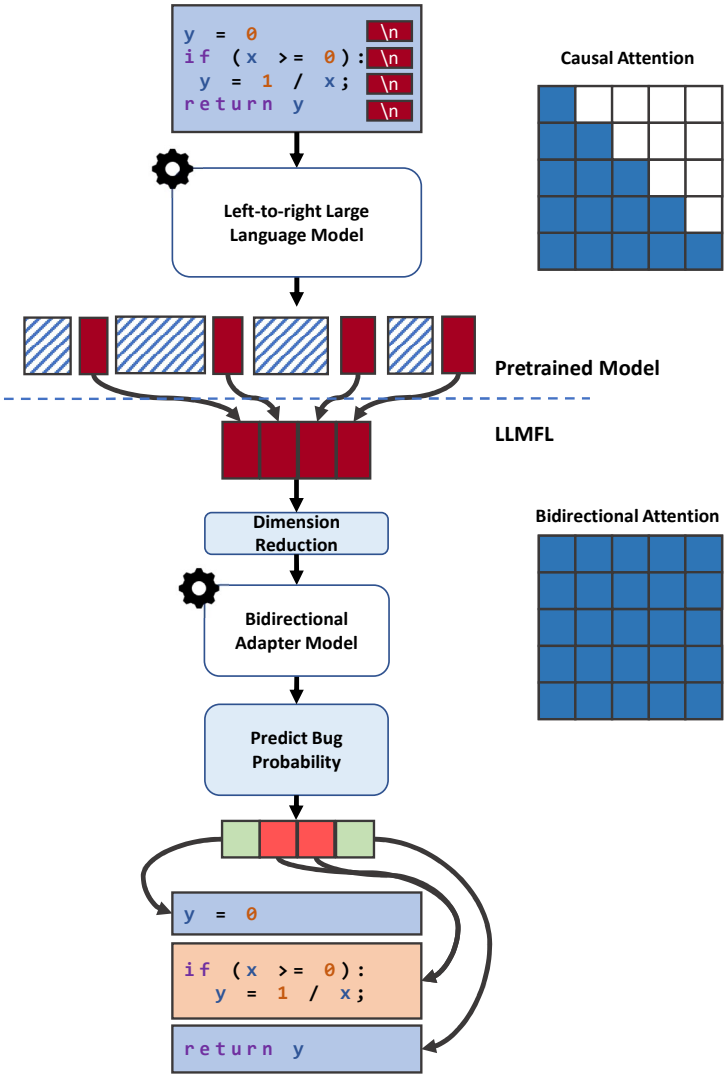

Figure 3.2: Attention masking procedure of *LLMAO*

Step 3: The dimension of the pretrained model's states, D, ranges up to 6,144 for the CodeGen models I built on. We use a significantly smaller dimension $d \ll D$ for our adapter layers, because they are trained on limited data. We 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. We 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. We 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 back-propagated through all layers up to, but not including, those in the pretrained network to obtain gradients. Given these gradients averaged across a sufficiently large mini-batch of samples, the model states are updated to make its predictions more likely to agree with the training labels.

3.3 Evaluation and Results

3.3.1 Dataset

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

- ***Defects4J* V1.2.0** : A Java benchmark dataset with 395 bugs from 6 Java projects [47]. I use V1.2.0 for most of our 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 [47]. To show that our approach can generalize to faults from unseen projects, we further evaluate our 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). We exclude the first 45 bugs in Jsoup and all in Gson/Jacksoncore because of trouble reproducing them (as seen in prior work [81]).
- ***BugsInPy***: a Python benchmark with 493 bugs from 17 different projects [131].
- ***Devign*:** a C benchmark with 5,260 from two open-source projects [149]. The original Devign dataset contains 15,512 security vulnerabilities from four different projects [149]. 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 [66, 87, 104]. I then track line numbers of changed statements as training labels.

3.3.2 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 our knowledge, no existing FL approaches take as input only the source code as natural language. However, we 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.

our baselines are recent, state-of-the-art statement-level MLFL approaches: DeepFL [65], DeepRL4FL [67], and TRANSFER-FL [87]. 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 [1], 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.

our 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 [65] is trained at the method level, i.e., predicting which methods are defective.

To compare, we 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, we directly cite the experimental results reported in their paper [67]. For each of the other compared techniques, we run their tool for a total time of 30 minutes, which is comparable to our tool's training time for 300 epochs.

3.3.3 Validation

For each of our three datasets, we perform a 10-fold cross validation on the entire dataset. Specifically, we 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) [65, 67, 81, 87]. 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 our 10-fold validation approach is more generalizable for training models on larger code datasets. As is done in some prior evaluations [65, 67], we also *separately* evaluate the degree to which our 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 our training runs. We 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 our model configurations), we stop training and use the best-performing checkpoint to calculate the Top-N metrics against the ground-truth labels.

3.3.4 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 [99], with particular attention paid to the top-5 elements [54]. To compare against state-of-the-art techniques, we adopt Top-N following prior work [65, 70, 81].

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

AUC of the model's ROC Curve. Although most developers inspect only top-5 elements in a given list, we 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 our model's top performing points in time, averaging precision and recall. We choose AUC to observe the prediction strength of our models at their peak performance.

3.3.5 Ablations

I conduct an ablation analysis to evaluate the impact of different components on the performance of our model (RQ3). I run five variants of our proposed technique for the *Defects4J* V1.2.0 dataset. I first evaluate *LLMAO* pretrained on *CodeGen*, and *LLMAO* without any pretraining to evaluate the impact of the pretrained large language model's final hidden states. For the pretrained models, we 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 our model without bidirectional layers, using only the *CodeGen* auto-regressive attention mechanisms for fault localization. We aim to determine if left-to-right LLMs can detect faults directly, without any customization for code understanding.

3.3.6 Hyperparameters

Table 3.1 shows the hyperparameters used in training all our 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 [43], we 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, we use a uniform batch size of 32 and perform gradient accumulation to ensure every batch of our data fits on a single GPU. For a fair comparison of *LLMAO*'s components (RQ3), we use the same number of training epochs (300) for all pretrain sizes and projected dimensions. However, the

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%)

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 our 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). We use a 8 attention heads for all our models.

3.3.7 Results

RQ1: How does *LLMAO* compare with prior DL-based FL tools? Table 3.2 (top) details experimental results showing how our tool compares against state-of-the-art FL techniques. The first 4 techniques are from prior approaches; I evaluate our *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. We observe that our 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 [132] 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, our approach is only slightly better than TRANSFER-FL, which performs roughly on-par with our *CodeGen*-6B model. However, note that prior techniques only achieve comparable results with our 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 [65]).

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 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 [81] 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 our approach is that *LLMAO* generalizes well to unseen projects *without retraining of any kind*. This argues for our 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.

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%)

Without Pretraining We trained our bidirectional language model from scratch, using the same tokenizer as *CodeGen* for tokenizing the inputs. We replace *CodeGen*'s token-level representation with a learnable embedding for each token. We 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 [132] 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 our tool on different pretrained *CodeGen* sizes, we 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.

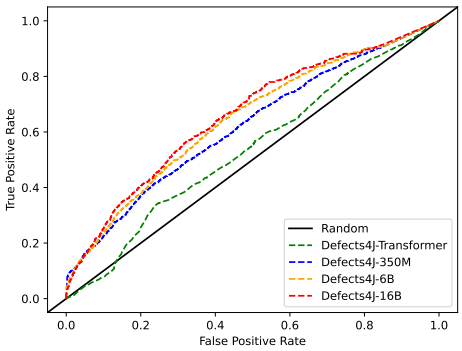

Figure 3.3: ROC curves on the *Defects4J* dataset

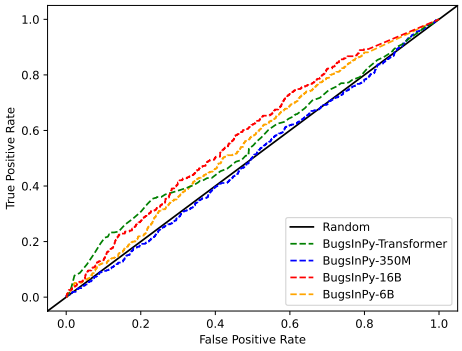

Figure 3.4: ROC curves on the *BugsInPy* dataset

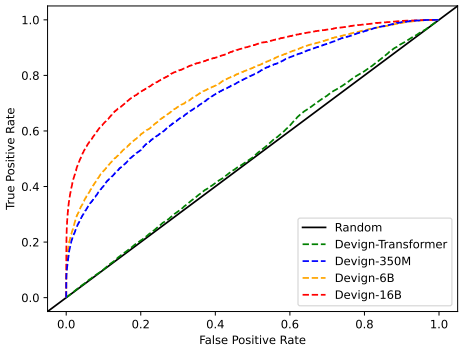

Figure 3.5: ROC curves on the *Devign* dataset

Figure 3.6: ROC curves on the completely random prediction, our 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.

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, we show that our tool using the largest pretrained LLM (i.e., *CodeGen* 16B) significantly outperforms all other variations of our model.

RQ4. How generalizable is *LLMAO* to other languages and domains? To evaluate our proposed technique on different languages and domains, we run all three pretrain sizes of our tool on the *BugsInPy* [131] dataset for localizing Python bugs, and the *Devign* [149] dataset for localizing C security vulnerabilities. We believe that measuring our 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*, we find that in total, *Defects4J* has 53% more code lines combined from all code files than the *BugsInPy* dataset. Since our approach considers source code as natural language, a larger database of code lines gives our models more training data. In particular, our largest dataset *Devign* with over 7 million lines of code achieves a Top-5 of 60.3% (i.e., 60.3% of our 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 our trained models compared to the completely random curve (i.e., AUC=0.5). A ROC curve shows the performance of our 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 our 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 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 our model's predictive power as I use a larger dataset of code corpus. *LLMAO* with *CodeGen*-16B trained on our smallest dataset *BugsInPy* yields an AUC of 0.571, and *LLMAO* with *CodeGen*-16B trained on our largest dataset *Devign* yields an AUC 0.855. We observe that our model's predictive performance on *Devign* is better than our 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 pretrained model scale up. *LLMAO* is also particularly effective for locating security bugs in C where test cases are not available.

3.4 Conclusion

This chapter presented LLMAO, the first test-free fault localization technique using bidirectional adapter fine-tuning on top of left-to-right language models. Unlike prior deep learning approaches (e.g., GRACE, DeepFL, DEAR, TransferFL) that require test coverage information and AST/CFG representations, LLMAO operates directly on source code without any test execution or complex preprocessing. This represents a fundamental departure from spectrum-based fault localization (SBFL) methods that rely on the ratio of passed and failed tests.

We evaluated LLMAO on three diverse benchmarks: 395 real bugs from Defects4J, 493 bugs from BugsInPy, and 5,260 security vulnerabilities from Devign. Our results demonstrate that LLMAO outperforms all prior state-of-the-art deep learning fault localization techniques across all three datasets. The key novelty lies in our bidirectional adapter architecture that enables discriminative tasks on causal language models, requiring only a few hundred training samples while leveraging the knowledge already encoded in pretrained LLMs.

This work directly supports my thesis by demonstrating that program analysis complements LLMs through structured signals—specifically, bidirectional code context that considers both prefix and suffix. By integrating this analysis-derived signal with LLM representations beyond autoregressive generation, LLMAO consistently outperforms pure learning-based approaches, providing quantitative evidence that explicitly integrating analysis signals with LLMs improves software maintenance tasks.

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.

This chapter demonstrates how program analysis complements LLMs for automated program repair, but uniquely shows that LLM properties themselves can serve as structured signals. While typical program analysis provides external signals to LLMs, this work shows that LLM uncertainty (entropy) acts as an analysis signal that can guide traditional APR techniques at all three stages: fault localization, patch generation efficiency, and patch correctness assessment. This bidirectional integration—where LLMs both consume and produce analysis signals—supports the thesis that explicitly integrating analysis-derived signals with LLMs beyond autoregressive generation improves software maintenance.

In this chapter, we 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" [103]. The naturalness of code and unnaturalness of buggy code is now a well-established phenomenon [41, 103]. However, the bulk of prior research on this topic relied on relatively simple n-gram language models [11]. 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 [135, 145].

We 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 [135] (acknowledging the risk of training data leakage in any such experiment [5]). 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* [76] 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 [60, 85, 136], template instantiation [51, 76], or machine learning models [135]) repair *efficiency* is best approximated in terms of the number of patches that must be evaluated to find a (good) repair [77]. 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 generalize to the desired behavior beyond what is tested in the provided test suite [111]. 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 [38, 117, 145]. 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 [111, 145]. I show that entropy can rank correct patches 49% more effectively (in Top-1) than state-of-the-art patch ranker Shibboleth [38], without using any project-specific training data.

4.1 Background and Related Work

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. [102] analyzed the reported bugs of three generate-and-validate APR tools: GenProg [59], RSRepair [101], and AE [130] systems, to find that producing correct results on a validation test suite is not enough to ensure patch correctness. Smith et al. [111] 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. [146] proposed ODS (Overfitting Detection System), a novel system to statically classify overfitting patches. Xiong et al. [136] generated both execution traces of patched programs and new tests to assess the correctness of patches. Ghanbari et al. [38] used both the syntactic and semantic similarity between original code and proposed patch, and code coverage of passing tests to rank patches. Shibboleth [38] was able to rank the correct patch in Top-1 and Top-2 positions in 66% of their curated dataset. Tian et al. [117] 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 [27], CC2Vec [42], and Doc2Vec [58] yield similarity scores that, given a buggy code, substantially differ between correctly-patched code and incorrectly-patched one. Yang et al. [145] 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 [136].

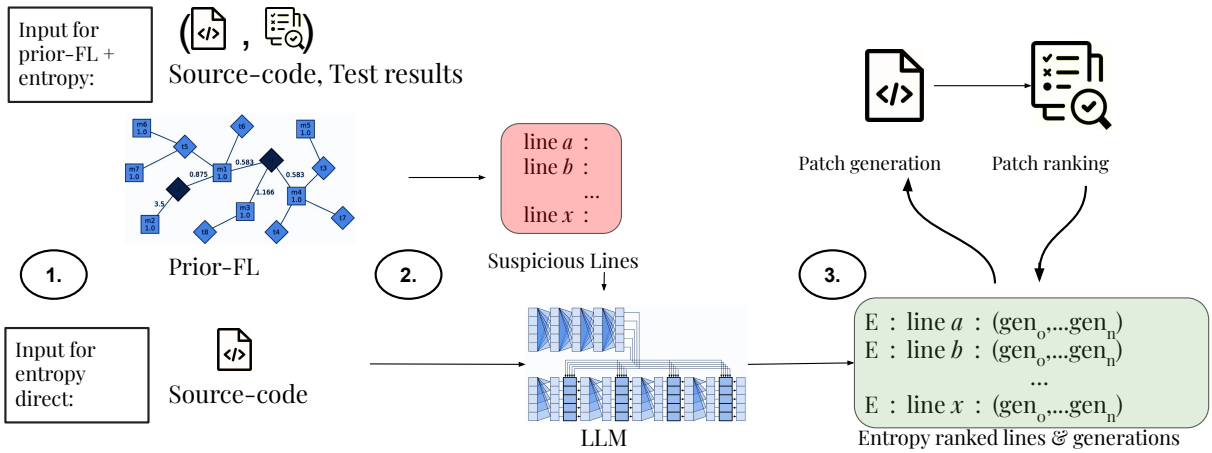

Figure 4.1: Fault localization pipeline using entropy. (1) I take a prior-FL suspicious score list, (2) query each code-line for entropy values, and (3) re-rank the list using LLM entropy scores.

4.2 LLM Entropy for APR

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

4.2.1 Entropy-Delta

To evaluate patch naturalness, which I use in both patch prioritization during generation/evaluation and patch correctness prediction, we 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 our 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, we query the entropy-delta between the "newline" token and the original line of code. For the case of an insertion, we 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.

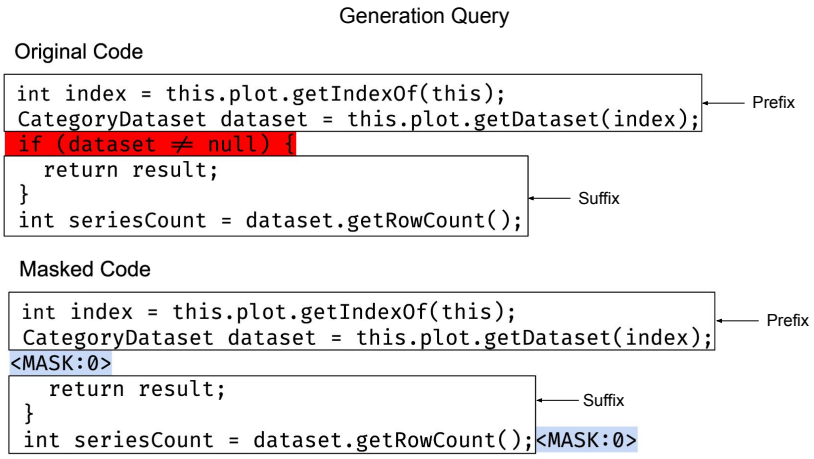

Figure 4.2: An example of entropy-delta query from a code-line deletion patch. The entropy-delta value of the deleted line is the difference between the original line and a blank line.

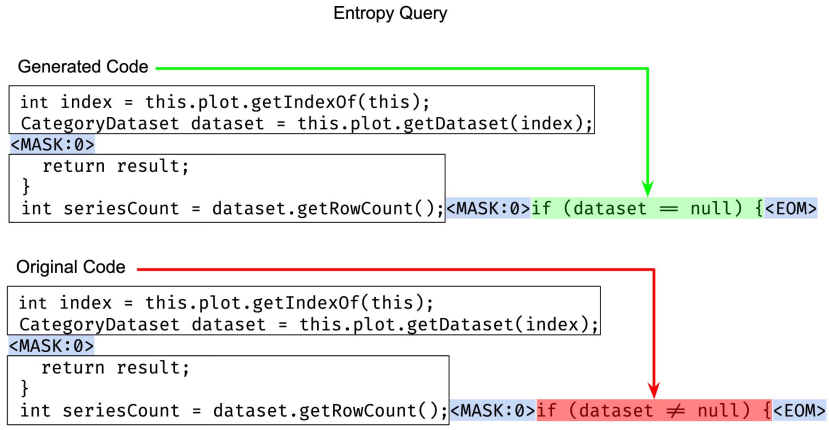

Figure 4.3: An example of entropy-delta query from a code-line replacement patch. The entropy-delta value of the replaced line is the difference between the original line and the replacement line.

4.2.2 Modified TBar

our patch efficiency experiments ask how entropy can speed up patch generation and evaluation. I evaluate it in context of TBar [76], 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 [69, 86, 135]) because our 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. Our experiments require several modifications to the codebase. First, we enable TBar to continue seeking patches after the first test-patching patch is found. Second, we enable TBar to generate patches, or evaluate them in a customized order (such as one provided by an entropy-delta ranking). our 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 our replication package.

4.3 Evaluation and Results

In this section, we describe the models I use for entropy (Section 4.3.1), the bug and patch datasets considered (Section 4.3.2), as well as evaluation metrics (Section 4.3.3).

4.3.1 LLMs

I used inCoder [34], Starcoder [61], and Code-Llama2 [118]. The three LLMs were trained on open-source code and are capable of infilling with bidirectional context. The inCoder model [34] 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 parameters. Code-Llama2 have three versions available: 7B, 13B and 34B. We 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, we set the LLM temperature to 0.5.

4.3.2 Dataset

I use the Defects4J [47] 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

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

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, we 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, we use curated datasets from prior tools' replication packages directly, namely, Shibboleth [38] and Panther [117]. 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 plausible patches on Defects4J V2.0 curated by Ghanbari et al. [38]. For patch classification, we use a dataset of 2,147 plausible patches on Defects4J V2.0 curated by Tian et al. [117]. The patches from Tian et al. [117] were generated by seven different APR techniques. Each bug in the dataset has one correct patch and several plausible (i.e., test passing) but incorrect ones. We 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, we only consider the 434 bugs in the 6 projects included by Shibboleth [38] and Panther [117] (shown in Table 4.1).

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

4.3.3 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 [77], and a proxy for compute time. **Patch correctness.** For patch classification tasks, we 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, we 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.

4.3.4 Results

RQ1. Can entropy improve patch generation efficiency?

In this section, we discuss the observed relationship of entropy and test-passing patches. We use entropy from inCoder. We 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 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, we 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. [77] 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.

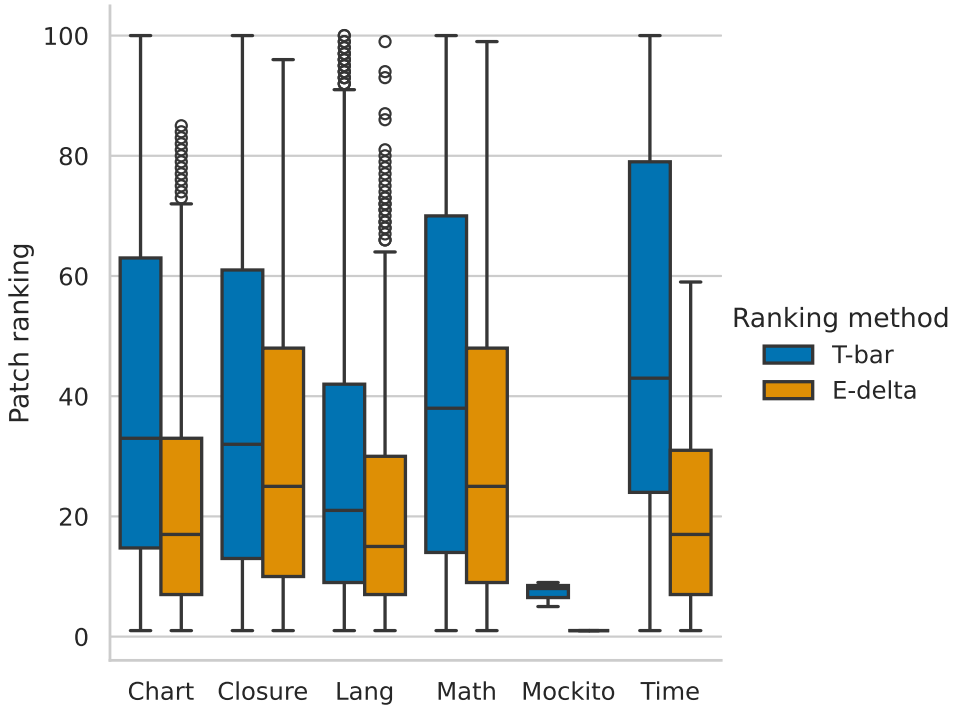

Figure 4.4: Entropy-delta and TBar ranking (lower is better) of test-passing patches on 72 Defects4J bugs.

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 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. We 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.

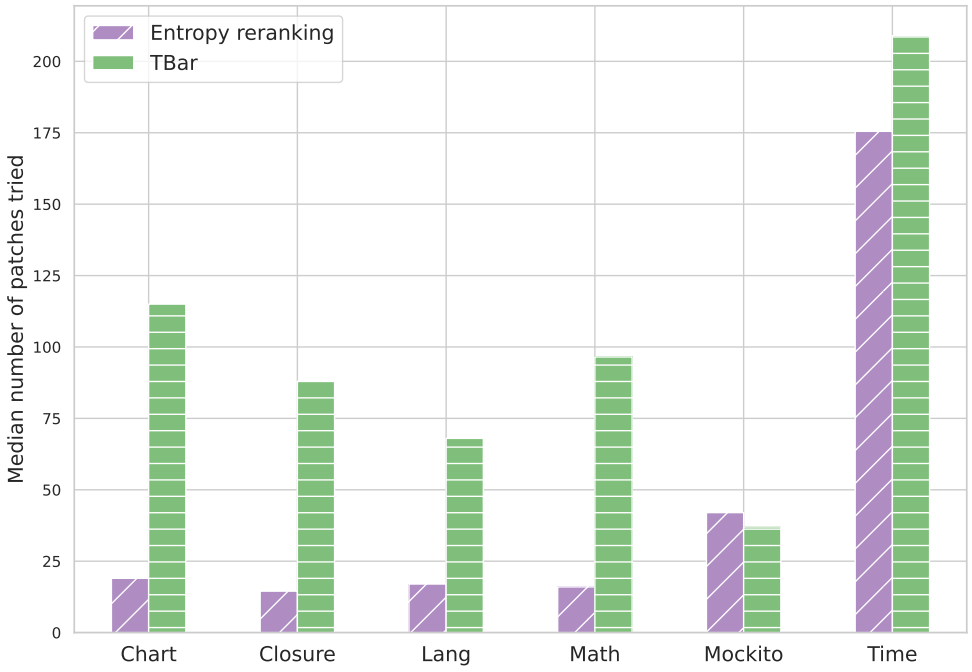

Figure 4.5: Median number of patches tested (lower is better) per project before successful patch using TBar original ranking and entropy-delta re-ranking of test-passing patches on 100 Defects4J bugs.

Finally, we 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 our experiment, 19 are identical to the developer fix.

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, we 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. [38]. For each bug, the dataset includes some number of plausible (i.e., test passing) patches,

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**

where exactly one is correct, and the rest are incorrect. We attempt to isolate the true correct patch from the incorrect patches. We 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 [38], we use Ochiai's SBFL for patch ranking as a baseline: ranking patches by the SBFL suspiciousness of the modified location.

Table 4.3 shows the Top-1 and Top-2 results of our approach on the labeled dataset of 1,290 patches. We see from Table 4.3 that entropy-delta outperforms both SBFL and Shibboleth [38] 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, we 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. We 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 our classification results on a labeled dataset of 2,147 plausible patches curated by Tian et al. [117] for classifying patches as correct or incorrect. Entropy-delta improves upon the accuracy score of PATCH-SIM [136] and Panther [117], but only slightly improves

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

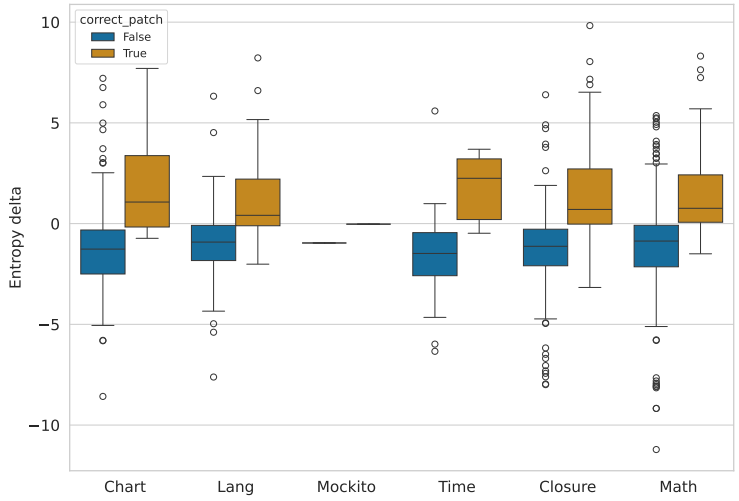

Figure 4.6: Entropy-delta across correct and incorrect patches on Defects4J projects. A higher entropy-delta signifies a less surprising patch to the LLM, and a lower entropy (sometimes negative) entropy-delta signifies a more surprising patch to the LLM.

+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

our 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, our results suggest that correct patches tend to lower entropy (i.e., increase naturalness) more than incor-

rect 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.

4.4 Conclusion

This chapter presented the first use of LLM entropy as a program analysis signal across all three stages of automated program repair: fault localization, patch generation efficiency, and patch correctness assessment. Unlike prior patch disambiguation tools (e.g., ODS, Patch-Sim, Shibboleth, Panther) that require project-specific training data, dynamic execution traces, or test coverage information, our entropy-based approach operates without any project-specific training and avoids the overfitting problems identified by Smith et al. [111] and Yang et al. [145].

We introduced entropy-delta, which measures the difference in entropy between a proposed patch and the original code, quantifying LLM uncertainty as a naturalness metric. Our evaluation on Defects4J demonstrates three key contributions: (1) entropy-delta saves an average of 24 test runs per bug for the template-based APR technique TBar, significantly improving repair efficiency; (2) entropy-based ranking improves Top-1 patch correctness by 49% compared to state-of-the-art patch ranker Shibboleth (which achieved 66% Top-1/Top-2 accuracy); and (3) entropy-based classification achieves 18% higher precision than prior patch disambiguation tools.

This work supports my thesis by demonstrating that program analysis complements LLMs through structured signals, including LLM-derived properties like uncertainty. By treating entropy as an analysis signal that can be integrated with traditional APR beyond autoregressive generation, this work shows that explicitly integrating analysis-derived signals with LLMs consistently outperforms pure learning-based or pure analysis-based approaches, providing quantitative evidence across all three APR stages.

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.

This chapter demonstrates how program analysis complements LLMs for code transpilation by providing formal verification of code properties as a structured signal. While autoregressive LLMs can generate idiomatic code, they provide no correctness guarantees. This work shows that formal verification—specifically equivalence checking via WebAssembly—provides essential analysis signals that validate LLM outputs beyond generation. By explicitly integrating verification-derived signals with LLM-generated code, this approach achieves both idiomaticity and functional correctness, supporting the thesis that analysis-derived signals complement LLMs for software maintenance.

In this chapter, we design, implement, and evaluate a LLM-based transpilation tool, and show that it is more idiomatic than prior non-LLM transpilers. Specifically, we use the Rust programming language as a case-study for our 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, we 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 Background and Related Work

5.1.1 Equivalence Verification

Equivalence verification has been studied in settings where two copies of should-be-equivalent artifacts are available: compiler optimizations [17, 18] and program verification [4, 56]. Our work uses equivalence verification for program language translation.

5.1.2 Rust Transpilers

Prior Rust transpilers convert C/C++ to Rust. C2Rust [120] automatically converts large-scale C programs to Rust while preserving C semantics. Citrus [122] and Bindgen [121] both generate Rust FFI bindings from C libraries, and produce Rust code without preserving C semantics. Bosamiya et al. [10] embedded WebAssembly (Wasm) semantics in safe Rust code for the Rust compiler to emit safe and executable Rust code. Bosamiya et al. [10] 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. [29] 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. [148] 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. [148] 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. [74] removed non-essential "unsafe" keywords in Rust function signatures and refined the scopes within unsafe block scopes to safe functions using code structure transformation. Our 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.

5.2 VERT: Verified Equivalent Rust Transpilation with LLMs

In this section, we 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, we 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, we 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, we describe each component of VERT in further detail.

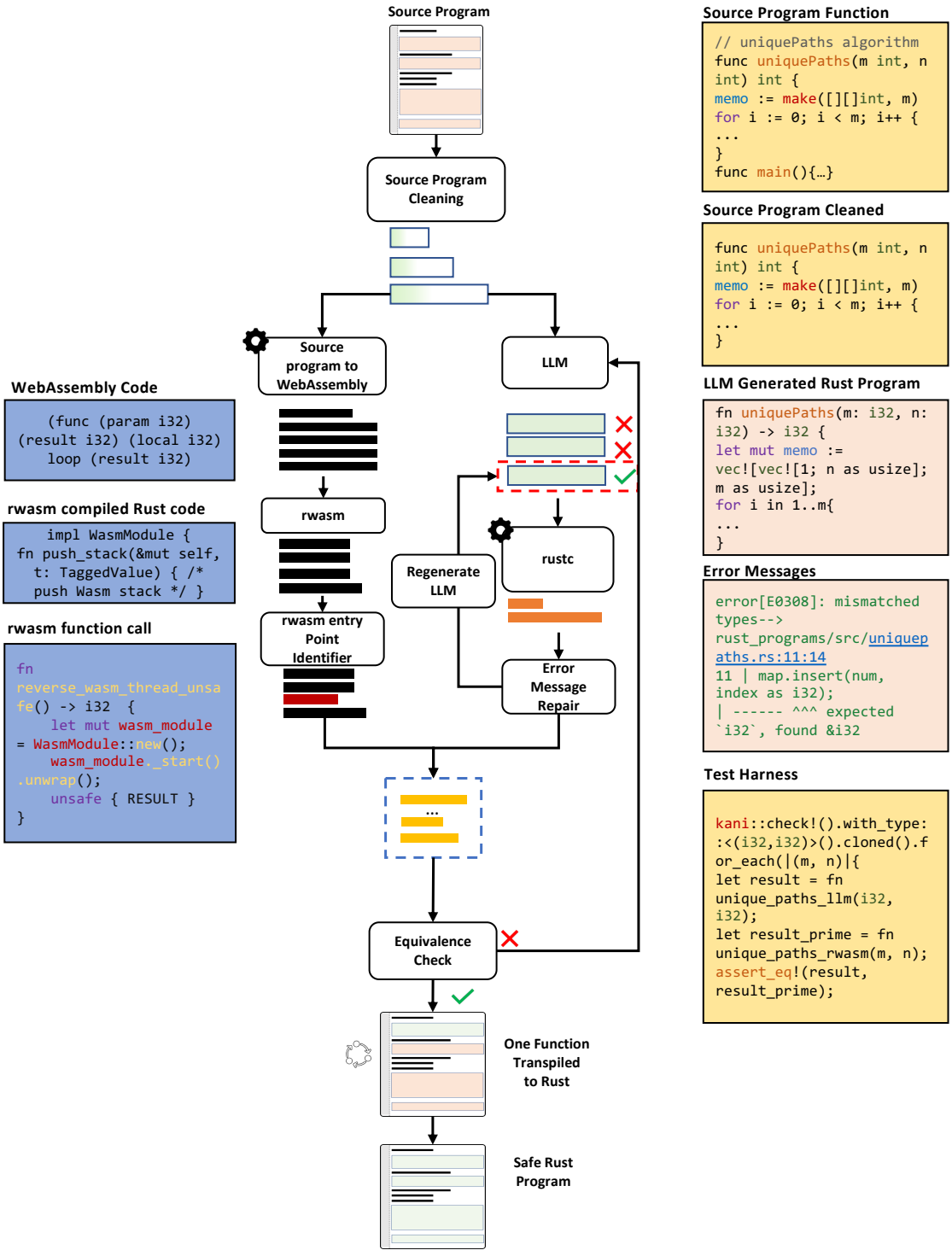

Figure 5.1: VERT's architecture, which takes as input a source program and produces a formally equivalent Rust program

```
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

5.2.1 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. We create an automatic repair system based on **`rustc`** error messages. For each error, we 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. We track the error code location (e.g., line 10 in Figure 5.2), and I use the **`rustc`** provided initial delimiter to guide our repair strategy. For this case, we 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`**), we 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, we replace **`x`** by **`mut x`**, and observe that the program compiles. It is often the case that even with the error message suggestion line, we cannot generate a repair that fixes all errors. In these cases, we regenerate a LLM output using the error as part of the new prompt and restart the process. Our error-guided repair is significantly faster than the LLM generation (discussed in Section 5.4), so I only regenerate a LLM output after exhausting all **`rustc`** helper messages.

```
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

5.2.2 Transpilation Oracle Generation

Since the LLM output cannot be trusted on its own, we create an alternate trusted transpilation pipeline for generating a reference Rust program against which the LLM output is checked. The 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. We 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, we use rWasm [10], 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, we use it to generate trusted Rust code with same semantics as the original input.

5.2.3 Mutation Guided Entry Point Identification

Given the assembly-like output of rWasm, we 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. We 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, we find that a mutation-guided approach is sufficient for our purposes. In Fig. 5.5, we know that the input is **`123`** and the output is **`321`**. Now, we 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, we guide this identification by leveraging the

```
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).

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, we can identify inputs and outputs by what changed. The diff in the rWasm output is shown in Fig. 5.6.

5.2.4 Equivalence Harness Generation

In our final step, we generate harnesses to check for equivalence given the input and output locations. We define equivalence here in functional terms: for all inputs, running both functions yields no crashes and identical outputs. To check this property holds, we 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, we leverage property-based testing with random inputs and model-checking with symbolic inputs. For the remainder of this section, we refer to both of them together as "the input."

The wrapper consists of two parts: input injection, and output checking. We replace constants inputs with the inputs of the harness like **`input: i32`**. Instead of replacing the parameters to the function, we 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, we 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. We 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.

```
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`.

5.2.5 Equivalence Checking

With the equivalence checking harness built, we 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. We use Bolero, a Rust testing and verification framework that can check properties using both Property-Based Testing (PBT) [32] and Bounded Model Checking (BMC) [19, 20].

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, we 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, we now perform bounded verification with a combination of Bolero and Kani. I customize Bolero to perform model checking through Kani [124], a model-checker for Rust. When run with Kani, Bolero produces symbolic inputs rather than random concrete inputs. Executing the harness with symbolic inputs, we 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. We 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. We 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), we measure that the median time to exhaustively verify a Rust program within our selected dataset is 5 minutes, which is much longer than our established time limit of 120 seconds per program. We choose not to use full Kani verification as a metric due to time and compute constraints.

```
{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.

I support complex types through their primitive parts. Given a struct or enum, that Kani or Bolero does not initially support, we 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, we 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. Our 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.

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

5.3 Evaluation Setup

In this section, we present our evaluation setup for the following research questions.

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

5.3.1 LLM Fine-tuning

The availability of Rust code in open source is scarce as compared code written in most other programming languages. Incoder [33] 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, we opt to not train a LLM targeted at Rust code generation. Instead, we directly use an off-the-shelf industry grade LLM, and also fine-tune on a separate open-source pretrained LLM. Specifically, we use Anthropic Claude-2 [2] for the industry grade LLM, and StarCoder [62] for the pretrained LLM.

I use light weight and parameter efficient adapter layers [45, 75, 140] 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 four languages, we find that LeetCode has the most consistent translation between other languages and Rust. We were able to collect 94 LeetCode questions of which have a direct translation between all 3 languages.

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

5.3.2 Benchmark selection

I draw our benchmarks from two sources. Our first source is the benchmark set from TransCoder-IR [115], 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. We choose this dataset to avoid potential data-leakage (i.e., LLM memorization) [7] in our evaluation. We note that the Rust programs produced by TransCoder-IR were released after June 2022, which is the training data cutoff date of our chosen LLMs [62, 107, 123]. I select programs from the TransCoder-IR dataset that can directly compile to Wasm using rWasm. After filtering, we 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.

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

To provide insight into VERT's ability to write safe rust, we gather 14 additional pointer-manipulating C programs from prior work on C to Rust transpilation [29, 120, 148]. We note, however, that the benchmarks in these prior works use open-source programs written before our chosen LLM's training data cutoff (June 2022). To avoid LLM data-leakage, we select and customize snippets from these C projects to transpile to Rust. We 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 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. We present several complexity metrics for the benchmarks and discuss them in more detail in Section 5.4.

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

5.3.3 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 runtime behavior. Conversely, two programs that share very few tokens (and hence have a very low text similarity score) could have identical compilation or runtime behavior. For programming languages, metrics based off of passing tests have been proposed. Roziere et al. [106] and Szafraniec et al. [115] 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, we use formal methods to measure the correctness of the output. In particular, we use property based testing (PBT) and bounded model checking (BMC). We 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, we set a time limit for our metrics. For both metrics, we 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.

5.4 Results

I present results on the TransCoder-IR benchmarks in Table 5.1. We 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, we apply the syntactic repair described in Section 5.2.1 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 5.2.6. *few-shot counter examples* only works for instruction-tuned models. We re-prompt the LLM up to 20 times for few-shot modes. For each LLM and each mode of VERT, we report the number of transpilations that compiled and that passed the various verification modes. As seen in Table 5.1, we only perform *single-shot* for Transcoder-IR (baseline) to replicate results from prior work. We 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, we 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 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. We 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 [53], we see that light-weight fine-tuning on a small dataset shows immediate improvement. In particular, we 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, we observe that an industry-grade LLM with more trainable parameters and a larger training dataset performs significantly better for our metrics.

I observe that VERT using few-shot with either StarCoder fine-tuned or Claude-2 yields

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**

better transpilation across all our 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. We find that the few-shot prompting for Claude-2 yields a greater improvement over single-shot compared to our 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. We observe that few-shot learning with counter examples of failed previous verification attempts provides the largest improvements on BMC. Modern LLMs

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

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 our entire evaluation dataset. We 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

our 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, we observe that VERT spends most of its runtime in verification.

5.5 Conclusion

This chapter presented VERT, the first verified Rust transpilation tool that combines LLM-based code generation with formal equivalence verification via WebAssembly. Unlike prior rule-based transpilers (e.g., C2Rust, Citrus, Bindgen) that produce unidiomatic or unsafe Rust code, and unlike pure LLM-based approaches that provide no correctness guarantees, VERT achieves both idiomaticity and functional correctness. Our approach is language-agnostic, requiring only a WebAssembly compiler for the source language, making it more general than prior work (e.g., CROWN, Emre et al.) that depends on source language memory management properties.

We evaluated VERT by transpiling 1,394 programs from C++, C, and Go to Rust. Using

the Claude LLM with bounded model checking verification, VERT successfully verified 40% more C++ programs, 37% more C programs, and 47% more Go programs compared to baseline approaches. This represents a significant improvement in achieving 100% functional correctness while maintaining code idiomaticity, a combination that neither rule-based transpilers (which sacrifice idiomaticity) nor pure LLM approaches (which sacrifice correctness) achieve alone.

This work supports my thesis by demonstrating that program analysis complements LLMs through structured signals—specifically, formal verification of code properties. By explicitly integrating verification-derived signals (equivalence checking) with LLM-generated code beyond autoregressive generation, VERT consistently outperforms pure learning-based or pure analysis-based approaches, providing quantitative evidence that analysis signals enable correctness guarantees for LLM-based software maintenance tasks.

6 Security Vulnerability Detection with Self-Instruct LLM Finetuning

Software security vulnerabilities allow attackers to perform malicious activities to disrupt software operations (i.e., security exploits). Since security exploits often occur during runtime, the independence of tests for LLM-based vulnerability detection shows promise.

This chapter demonstrates how program analysis complements LLMs for security vulnerability detection by providing dataflow properties as structured signals. While Chapter 3 used bidirectional context for single-function fault localization, security vulnerabilities often span multiple functions and files, requiring dataflow analysis across entire codebases. This work shows that integrating dataflow properties through graph neural networks with LLM fine-tuning beyond autoregressive generation enables effective vulnerability detection. This approach supports the thesis by demonstrating that explicitly integrating analysis-derived signals (dataflow graphs) with LLMs outperforms pure learning-based or pure analysis-based approaches for complex software maintenance tasks.

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.

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. However, a key limitation of LLMAO is the size of potentially vulnerable programs is limited by the maximum size of LLM context windows.

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*.

To circumvent the LLM maximum context window issue while expanding our target programs for vulnerability detection, we trained LLMs using small snippets of code accompanied by important and relevant information of the code snippet, inferred by static analysis vulnerability explanations. As the final thrust of our dissertation work, we expanded our prior work to detecting vulnerabilities from larger programs, by detecting vulnerable functions across entire

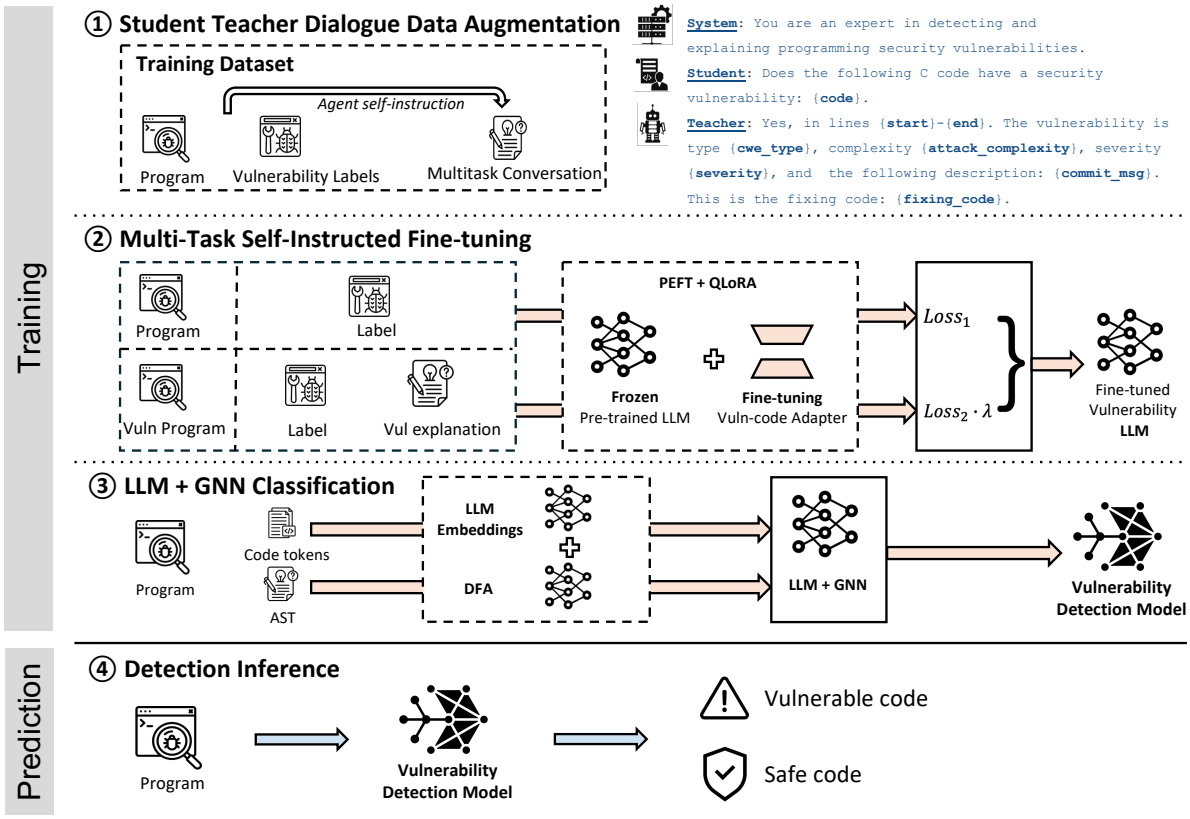

Figure 6.1: *MSIVD*'s architecture, which takes as training data a code snippet, its vulnerability label, and various human annotated vulnerability labels. *MSIVD* outputs a final vulnerability classification on unseen code snippets.

repositories instead of lines of code from single functions (i.e., LLMAO). We used static analysis of a potentially vulnerable program into trainable objectives for LLM-based vulnerability detection, and train LLMs on multiple dimensions of vulnerability information (i.e., multitask learning) 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. We extract a subset of previously established security vulnerability datasets BigVul and PreciseBugs for model evaluation, and perform ablation studies on different combinations of pre-trained LLMs and security vulnerability explanations. Finally I propose our tool *MSIVD*: a multitask self-instruction LLM model for security vulnerability detection that trains on multiple types of vulnerability information.

6.1 Background and Related Work

Security vulnerability detection has a rich research history covering both static and dynamic techniques (see., e.g., ref [83, 110, 116]). For example, code-similarity-based methods [52, 72] can detect vulnerabilities incurred by code cloning, and pattern-based methods [93, 138] use rules to identify vulnerabilities based on code patterns. However, code-similarity techniques show high false negative rates (many vulnerabilities are not caused by cloning), and pattern-based techniques require human experts to define the initial patterns. We focus on work using machine learning for static vulnerability detection, as it is most closely related to ours, requires less human expert intervention, and recent results are especially promising.

Devign [149] and IVDetect [68] used GNN on a program's AST or CFG to learn a program's likelihood to be vulnerable. LineVul [40] queried the final attention layers of a language model (CodeBERT) for the specific purpose of vulnerability detection. DeepDFA [114] uses DFA algorithms to train a GNN and achieve state-of-the-art vulnerability detection results more efficiently than prior program analysis based tools. Steenhoek et al. [114] observed that DeepDFA, when combined with LineVul [40], yields a higher vulnerability detection effectiveness than all prior Transformer-based tools.

6.2 MSIVD

Figure 6.1 provides an overview of MSIVD. The first three phases constitute training; the fourth phase, inference/vulnerability detection. The design rationale behind *MSIVD* is multi-faceted.

First, because security vulnerabilities are fundamentally complex, vulnerability reports and curated datasets include multi-dimensional information, e.g., textual categorization and explanations, implicated code, potential fixes, and static analysis reports. I aim to leverage these diverse types of *rich data* for training. During ① self-instruct dialogue-based data augmentation, *MSIVD* prepares a given dataset for fine-tuning by extracting vulnerability characteristics including type, description, and source code location. The second step, ② multi-task fine-tuning, then uses multi-task learning to fine-tune an LLM towards two learning objectives: (1) detecting a vulnerability and (2) providing an explanation that describes its characteristics. We focus on using *MSIVD* for vulnerability detection, and leave an evaluation (i.e., a human study) of the quality of produced explanations to future work. However, we predicate *MSIVD* on this type of multitask learning, including the explanation objective, because doing so can allow a model to simultaneously learn shared knowledge and patterns, improving generalization and accuracy. I overcome the small size of the available security vulnerability datasets by taking advantage of recent LLM advances that enable lightweight, parameter efficient [75], and quantized adapter level fine-tuning [139] suitable for smaller training datasets. Although these first two steps incorporate some code-level information, security vulnerabilities often entail specific, non-local, and subtle information flow. To reason about deeper semantic information, ③ LLM+GNN training, jointly trains the LLM with a GNN based on information flow data derived from the program's control flow graph. This step represents the program as a graph, allowing the fine-tuned LLM to reason directly about whether the values and data relations in a program indicate the occurrence of a vulnerability. In the ④ detection phase, given a program, the vulnerability detection LLM

trained by *MSIVD* predicts whether a program contains a vulnerability.

6.2.1 Data Augmentation

To achieve security vulnerability detection incorporating vulnerability explanations, we first curated a custom dataset for both training and evaluating. The original dataset I use include samples of code snippets, and manually labelled classification on whether or not it includes a security vulnerability. The augmented dataset must allow the LLM to perform chain-of-thought, reasoning based on Self-instruct and Dialogue-policy-planned training. Therefore, the datapoints in our augmented vulnerability dataset must include a code snippet, its associated vulnerability label, CWE-type, a vulnerability description (e.g., how an attacker could exploit the vulnerability), and developer fix with fix location.

I transform a classification based security vulnerability dataset into a suitable dialogue, enabling a multi-round conversation format between a teacher and student. Inserting intermediate reasoning steps like this improves the ability of an LLM to perform complex reasoning. Each complete dialogue is a single training data entry. Embedded with the conversation is first a system prompt asserting that the teacher is "an expert in detecting and explaining programming security vulnerabilities", followed by a back-and-forth of questions and answers about the vulnerability and its associated information. Figure 6.2 shows a complete dialogue training data entry example. The teacher and student converse in three rounds of dialogue, each on a different aspect of the security vulnerability in a target code snippet. The first round of dialogue discusses the existence of the vulnerability; the second round, an explanation of why the code snippet has a vulnerability; and the third, which lines need to be changed to fix the vulnerability.

6.2.2 Model setup

I train two models for *MSIVD*. The first performs a sequence-to-sequence fine-tuning of a selected pre-trained LLM, using our multitask self-instruct approach. The second performs a sequence-to-classification training loop that outputs a binary classification label (i.e., if the sample is vulnerable or not), which we build on top of DeepDFA's GNN architecture. The second model takes the final hidden states from the frozen in place first model. We refer to the tool using both models as *MSIVD* throughout evaluation; the tool consisting only of the first model, without the GNN architecture, as *MSIVD-*. *MSIVD-* converts the first model into a sequence-to-classification model directly, using a single `linear` layer. For the initial pre-trained model, we use CodeLlama-13B-Instruct [107], which is the 13 billion parameters instruction tuned version of CodeLlama. CodeLlama released 4 model size versions, from 7B to 70B. Due to limited computing and VRAM, we chose the 13B version. Table 6.1 shows the hyperparameters used for both models. The 4352 model dimension from the LLM-GNN model is a result of concatenating the fine-tuned LLM (4096) with the GNN model (256). Similarly, we add the output layers of LLM with the GNN to form $8 + 3 = 11$ layers. For batch size, we use 4 to fit CodeLlama 13B onto a single RTX 8000 GPU. However, other GPUs with more VRAM could employ higher batch sizes for greater efficiency. All experients used an Intel(R) Xeon(R) 6248R CPU @ 3.00GHz running Debian GNU/Linux 1 and two Nvidia Quadro RTX 8000 GPUs.

```
Round 0 = {
  role: ``System'',
  content: "You are an expert in detecting and locating security
      vulnerabilities, and can
    help answer vulnerability questions",
},
Round 1 = {
  role = ``Student'',
  content = f"Does the following code have any security vulnerabilities
      : {code_snippet}",

  role = ``Teacher'',
  content = f"Yes. The following code has a vulnerability type {
      cwe_type}.",
},
Round 2 = {
  role = ``Student'',
  content = f"What is the vulnerability description?",

  role = ``Teacher'',
  content =f"The vulnerability is:{commit_msg}",
}
Round 3 = {
  role = ``Student'',
  content = f"Locate the lines that are vulnerable and should be
      repaired.",

  role = ``Teacher'',
  content = f"The code is vulnerable at lines {vuln_lines}, with the
      following fix: {fixing_code}",
}
```

Figure 6.2: A single training data entry for the vulnerability detection multi-task fine-tuning. The dialogue rounds follow the four types of labelled data: vulnerability classification label, description, type, and repair lines.

6.3 Evaluation and Results

In this section, we present the results evaluating *MSIVD*'s performance by answering three research questions:

RQ1: How effective is *MSIVD* at finding vulnerabilities on an established dataset?

RQ2: To what extent can *MSIVD* generalize to known-unseen vulnerabilities?

RQ3: How does each component of *MSIVD*, or vulnerability type, impact performance?

All results presented in this section were obtained using an Intel(R) Xeon(R) 6248R CPU @ 3.00GHz running Debian GNU/Linux 1 and two Nvidia Quadro RTX 8000 GPUs.

Table 6.1: Hyperparameters for multitask self-instruct fine-tuning, and LLM-GNN combined vulnerability detection model training.

Hyperparameter	Multitask FT	LLM+GNN
Initial Learning Rate	1e-5	1e-6
Model Dimension	4096	4352
Context Window	2048	2048
Layers	8	11
Batch Size	4	4
Epochs	10	5

6.3.1 Datasets

Established Dataset

Recently-proposed vulnerability detection models [35, 40, 68, 114] are typically evaluated on the *Devign* [149] or *BigVul* [30] datasets of real-world C/C++ projects and vulnerabilities. We choose *BigVul* for our evaluation because it is equipped with both labelled code snippets (functions) and CWE explanations (*Devign* only contains labeled code snippets); is an order of magnitude larger; and is imbalanced. Following prior work, and for fair comparison with IVDetect [68], LineVul [35] and DeepDFA [114], we split *BigVul* into a 80/10/10 split on training, evaluating, and testing. We exclude the same 1,564 labeled functions (0.8%) from the dataset as LineVul [35] and DeepDFA [114], namely those with functions that (1) are incomplete (i.e., ending with ');', or not ending in '}') that cannot be parsed, (2) contain no added nor removed lines, but were simply labelled vulnerable. (3) entailed a fix that modified more than 70% of lines, indicating a substantial change that may fundamentally change the vulnerable code, or (4) are fewer than 5 lines long.

Novel Dataset

BigVul contains vulnerabilities sampled from *before* most modern LLM's training cut-off date of January 2023 [30, 35, 149]. Since our tool is based on pre-trained LLMs, we also collect labelled vulnerability data produced after that date, to mitigate the risk of data leakage. We use the *PreciseBugCollector* [39] toolset to produce this dataset. *PreciseBugCollector* mines verified vulnerabilities reported by human annotators from the National Vulnerability Dataset (NVD), which includes significant and well-known vulnerabilities, such as HeartBleed (CVE-2014-0160[1]) and Log4Shell (CVE-2021-44228[2]). *PreciseBugCollector* uses the public NVD API[3] to download comprehensive vulnerability metadata for human-expert-confirmed vulnerabilities. We collect the 2,543 vulnerabilities (of 217,403 overall) in the NVD database that contain external links to GitHub commits, tagged with *Patch*, touching C or C++ code (like *BigVul*), and extract the fixed source code accordingly. Splitting the 2,543 vulnerabilities into individual file patches produces

[1] `https://nvd.nist.gov/vuln/detail/cve-2014-0160`
[2] `https://nvd.nist.gov/vuln/detail/cve-2021-44228`
[3] `https://nvd.nist.gov/developers/vulnerabilities`

12,970 file-level changes.[4] We leave consideration of multi-language tuning and detection, a known challenge [147], to future work. Note that *PreciseBugCollector* also provides a fault injection mode that can create large quantities of synthetic defects; we do not use this mode, and collect only real reported bugs and associated metadata and explanations from NVD.

Our *PreciseBugs* dataset consists of 80% non-vulnerable and 20% vulnerable samples. Vulnerable samples are constructed from the human-patched code. We purposefully craft our dataset to be unbalanced to make it more difficult for a model to guess (i.e., 50% coin toss) the correct answer, and to replicate real world settings (most code in the real world is not vulnerable). We split the dataset into a 80/10/10 split on training, evaluating, and testing. We train on vulnerability samples from before January 2023, and produce evaluation and testing datasets of vulnerabilities where the associated vulnerability label and code fix occurred after January 2023. This both mitigates the risk of data leakage in the pre-trained LLMs, and represents realistic usage (a model will necessarily always train on previously-discovered vulnerabilities, to detect future, as-yet-undiscovered vulnerabilities).

Our augmentation of the original *PreciseBugs* dataset therefore entails (1) splitting samples into single file code snippets to fit within LLM context windows, (2) processing samples into a student-teacher dialogue format, and, most importantly, (3) re-running *PreciseBugCollector* on 2023–2024 labelled vulnerabilities to mitigate LLM evaluation data leakage.

6.3.2 Benchmarks

For classification, we convert the existence of a vulnerability into binary labels. To characterize classification effectiveness for an entire dataset, we use F1, precision, and recall, following the same metrics used for the prior work [35, 68, 73, 114].

Security vulnerability detection has a rich research history long pre-dating recent advances in machine learning. We focus our empirical comparison on work that uses machine learning for static vulnerability detection, as the recent results are especially promising, including compared to older, non-ML-based approaches. We compare *MSIVD* to techniques that target general vulnerability detection (that is, not restricted to particular rule-sets or vulnerability types) at the same granularity level (the function level),[5] and that either provide replication packages that can be applied to *BigVul*, or include evaluation results on *BigVul* in the original publication. Our chosen baselines cover several categories:

- ***Non-LLM** deep learning-based vulnerability detection tools* VulDeePecker [73], SySeVR [128], Draper [108], IVDetect [68], and DeepDFA [114].
- ***LLM**-based approaches*, including (1) the vulnerability detection capabilities of open-source, pre-trained LLM models CodeBERT [31], CodeT5 [128], and CodeLlama [107] (code pre-trained version of Llama-2), and (2) LineVul [35], which trains an additional sequence-classification model on top of a pre-trained LLM. Although LineVul originally uses CodeBERT and RoBERTA [78] as its pre-trained LLM, we customize it to use the same pre-trained

[4]Our tool, like prior work, assumes that vulnerabilities are localized to single files, which is not always the case; we discuss limitations to this assumption in Section **??**.

[5]This granularity choice excludes several recent techniques, like Vuddy [52], VulChcker [90], and LineVD [40]; these are statement-level.

Table 6.2: Vulnerability prediction effectiveness on the Bigvul dataset. VulDeePecker, SySeVR, Draper, and IVDetect performance are from ref [68], and CodeBERT and CodeT5 performance from ref [114].

Type	Technique	F1	Precision	Recall
Random	Random	0.11	0.06	0.50
Non-LLM	VulDeePecker	0.12	0.49	0.19
	SySeVR	0.15	0.74	0.27
	Draper	0.16	0.48	0.24
	IVDetect	0.23	0.72	0.35
	DeepDFA	0.67	0.54	0.90
LLM	CodeBERT	0.21	0.68	0.13
	CodeT5	0.46	0.56	0.39
	CodeLlama	0.74	0.85	0.63
	LineVul	0.81	0.86	0.78
LLM + GNN	CodeT5 + DeepDFA	0.79	0.85	0.71
	LineVul + DeepDFA	0.88	0.88	0.89
	MSIVD	**0.92**	**0.93**	**0.91**

model (CodeLlama-13B) as *MSIVD* and train for the same number of epochs (5). Otherwise, differences between *MSIVD* and LineVul could be a due to differences in pre-trained model effectiveness, rather than approach.

- ***LLM + GNN** combined techniques* using DeepDFA's replication package to combine their GNN embeddings with our fine-tuned model, as well as any HuggingFace pre-trained model directly. We train the same number of epochs (5) as *MSIVD*.
- ***Random*** baseline that predicts whether a sample is vulnerable with a probability of 0.5, to ground precision, recall, and F1, whose performance is sensitive to the underlying data distribution (and our datasets are imbalanced).

Our chosen baselines represent the state-of-the-art of vulnerability detection models [114], and all can be evaluated on the the *BigVul* dataset. Prior techniques relying on extant program analysis results cannot be trivially evaluated on the *PreciseBugs* dataset; we therefore only evaluate the best-performing LLM-based techniques on it.

6.3.3 Results

6.3.4 RQ1: Effectiveness on established dataset

Table 6.2 shows the effectiveness of *MSIVD*, and all baselines, on the *BigVul* dataset. DeepDFA's data flow analysis-based GNN technique outperforms prior non-LLM techniques, with a F1 score of 0.67, largely due to its high recall of 0.9 (correctly identifying 90% of the vulnerable code samples). However, Table 6.2 also shows that *all* LLM approaches other than CodeBERT outperform all non-LLM approaches. That is, LineVul, without any insights from program analy-

sis, surpasses all state-of-the-art program-analysis-based deep learning tools, although it is even more effective when combined with DeepDFA's GNN layers. Model knowledge from static analysis provides limited improvement for more powerful LLMs like CodeLlama; this knowledge more clearly benefits more dated LLMs like CodeT5 (i.e., F1 score improving from 0.46 to 0.79). Overall, *MSVID* achieves F1 of 0.92, precision 0.93, and recall 0.91, outperforming all baselines on all metrics. These results demonstrate that the different aspects of vulnerability explanation can indeed improve a pre-trained LLM's vulnerability detection accuracy. However, *MSVID* only shows incremental improvements over LineVul + DeepDFA, as compared to LineVul + DeepDFA's larger improvements over all non-LLM tools. The largest improvements on vulnerability detection with the *BigVul* dataset comes from the underlying LLM itself.

RQ1 Summary

LLM-based techniques outperform non-LLM techniques on the *bigvul* dataset. *MSVID* outperforms prior all state-of-the-art approaches, with an F1 score of 0.92. The incremental improvements of adding either GNNs or fine-tuning suggests that the underlying pre-trained LLM is capable of effective vulnerability prediction based on code tokens alone.

6.3.5 RQ2: *MSIVD* effectiveness on unseen vulnerabilities

Table 6.2 shows *MSIVD*-'s performance on the *PreciseBugs* dataset, comparing to LineVul (the best-performing prior technique on *BigVul*). *MSIVD* shows a larger improvement over LineVul on the newer dataset (F1 of 0.48 to LineVul's F1 of 0.31) as compared to on *BigVul*. This demonstrates the relative effectiveness of our fine-tuning approach on unseen vulnerabilities.

We sought to understand why LineVul with CodeLlama as the underlying model performs so much worse on *PreciseBugs* than on *BigVul*. One clue lies in CodeLlama's effectiveness on our evaluation data on its own: Table 6.2 shows that CodeLlama achieves an F1 score of 0.74 on *BigVul*, but only 0.22 on *PreciseBugs*. Without any additional vulnerability classification training or sequence-to-sequence fine-tuning, CodeLlama already beats most prior non-LLM techniques.

More telling evidence emerges from the training loss function for CodeLlama using our multitask fine-tuning method. A deep-learning model's loss curve describes how closely a model's predictions are to the ground truth. We expect loss to generally decrease as training epochs increase; lower loss means better prediction. Figure 6.3 shows the loss curves of our training approach on the *BigVul* dataset with (i.e., multitask fine-tuning) and without explanations (i.e., label-only fine-tuning). The loss curve on fine-tuning *BigVul* with explanation approaches 0.2 in 400 steps (2 epochs, roughly 16 hours of training time). In contrast, the loss curve on fine-tuning *BigVul* without explanations approaches 0.2 in 50 steps (1/4 of an epoch, roughly 2 hours of training time). Near-zero loss, with minimal training, is a strong indication of data leakage and overfitting. Here, the pre-trained LLM (CodeLlama) shows clear over-fitting (i.e., prior memorization of the dataset) on *BigVul* without explanations. Adding explanations includes more information, and adding non-binary labelling naturally induces noise, explaining why fine-tuning with explanations is not as overfit. Note that we observe the same loss curve behavior on the *Devign* dataset, using the same setup (not shown). This suggests that much of the accuracy of LLM-based techniques is likely arises from dataset memorization. This supports the importance of evaluating LLM-based approaches on labelled vulnerabilities collected after the

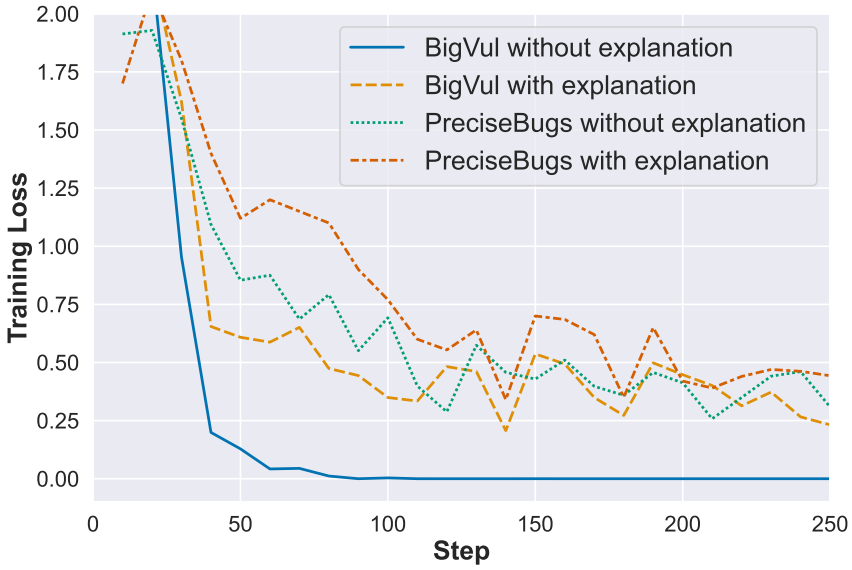

Figure 6.3: Loss curve of *MSIVD* fine-tuning CodeLlama for *BigVul* and *PreciseBugs*. Lower loss indicates model predictions closer to the ground-truth labels. Near-zero loss, with minimal training, indicates over-fitting [127]. Note that the pre-trained LLM (CodeLlama) shows obvious over-fitting on *BigVul* without explanations, motivating the necessity for new labelled datasets for model evaluation.

selected LLM's training data cut-off date. Our results also demonstrate the value of fine-tuning LLMs using a multitask approach which, even on previously seen data, inserts a useful degree of randomness in learning.

RQ2 Summary

Neither CodeLlama alone nor prior LLM-based vulnerability detector baselines generalize well to the unseen *PreciseBugs* dataset. Training loss curves suggest that CodeLlama has likely memorized the *BigVul* dataset. While *MSIVD* is therefore more effective on *BigVul* than on *PreciseBugs* vulnerabilities, it better generalizes to the recently released *PreciseBugs* dataset than the prior baselines.

6.3.6 RQ3: Ablation study

Setup. To answer RQ3, we evaluate *MSIVD* under four settings and evaluate their performances on *BigVul* and *PreciseBugs*. First, we use the underlying pre-trained LLM directly for prediction (as in *RQ1*). We then use a fine-tuned version of *MSIVD*, but without any vulnerability explanations in its training data (label-only FT). Finally, we include vulnerability explanations in a single round of self-instruction fine-tuning (single-round SIFT) and multiple rounds of self-instruction fine-tuning (multi-round SIFT, which corresponds to *MSIVD*-). For *BigVul*, we also add the GNN adapter layers (multi-round SIFT + GNN, which corresponds to the full version of *MSIVD*).

We additionally evaluate our tool on specific vulnerability types within *PreciseBugs*, (training/evaluating on single vulnerability types). We choose the three most common types from our *PreciseBugs* dataset: buffer error (27.3% of *PreciseBugs*), resource error (21.2% of *Precise-*

Bugs), and input validation error (13.6%).

Results. Table 6.3 shows results on both the *BigVul* and *PreciseBugs* datasets. As discussed in Section 6.3.5, CodeLlama already performs well at detecting *BigVul* vulnerabilities. Training a separate model without using agent self-instruction slightly improves effectiveness, with a F1 score of 0.81 (+0.07 above the pre-trained F1 score of 0.74) for *BigVul*, and a F1 score of 0.33 (+0.1 above pre-trained) on *PreciseBugs*.

Surprisingly, we find that fine-tuning on only the vulnerability label and none of the explanations actually performs worse than using a pre-trained model directly for vulnerability classification (0.71 F1 for fine-tuned CodeLlama, and 0.74 F1 for pre-trained CodeLlama on the *BigVul* dataset). Our findings are consistent with those of Yusuf et al. [147], who observed that instruction-based fine-tuning may not always enhance performance, especially across a dataset of diverse CWEs. The shift from sequence-to-sequence fine-tuning to the sequence-classification training within a small dataset may simply include more noise, reducing classification performance.

Fine-tuning with both code and vulnerability explanations with the multitask agent setup (*MSIVD*) yields the highest vulnerability detection on both *BigVul* and *PreciseBugs*. We also see that training with multi-round SIFT yields higher F1 scores than single-round SIFT (a F1 improvement of 0.09 for *BigVul*, and 0.02 for *PreciseBugs*), which is consistent with prior work on LLM instruction-prompting [55]. Finally, we observe that the additional GNN (multi-round SIFT + GNN) provides an additional 0.02 F1 on top of multi-round SIFT for the *BigVul* dataset. The incremental improvement from the addition of GNN shows that CodeLlama already makes accurate predictions based on prior knowledge on the *BigVul* dataset, as previously discussed in Section 6.3.5.

Table 6.3 shows that training and evaluating on single vulnerability types improves F1 scores as compared to the entire dataset, but by trading off higher precision for lower recall. These results further corroborate that the LLM-unseen vulnerabilities in the newer *PreciseBugs* dataset are more difficult for any language model to detect. However, our results also indicate that training with a multi-round self-instruct format on a dataset with both label and explanation produces considerable improvements over pre-trained models alone, and substantiate the value of the individual components of *MSIVD*'s design.

RQ3 Summary

Further training a code LLM on vulnerability-specific code and labels improves detection effectiveness. Fine-tuning an LLM without vulnerability explanations actually reduces effectiveness as compared to the pre-trained model alone. Multitask fine-tuning with all included vulnerability explanations achieves the highest detection effectiveness, especially with multiple rounds of self-instruction. Finally, selecting specific vulnerability types for both training and evaluating yields higher F1 scores, but with a trade-off of lower recall due to the smaller data size.

6.4 Conclusion

This chapter presented MSIVD, the first multi-task self-instruction LLM fine-tuning approach for security vulnerability detection that simultaneously learns vulnerability classification and ex-

Table 6.3: *MSIVD* ablation study. "Pre-trained" uses the underlying LLM directly for vulnerability detection. "Label-only fine-tuned" (FT) performs single-task fine-tuning on the vulnerability classification labels. "Single round self-instruct fine-tuned (SIFT)" trains the LLM without the agent explanation multi-round dialogue. "Multi-round SIFT" uses multi-task agent-dialogue fine-tuning (*MSIVD*$^{-}$). "Multi-round SIFT + GNN" adds the GNN adapter layer and corresponds to the full version of *MSIVD*.

Dataset	Technique	F1	Precision	Recall
BigVul	Pre-trained	0.74	0.85	0.55
	Label-only FT	0.71	0.77	0.66
	Single-round SIFT	0.81	0.86	0.61
	Multi-round SIFT	0.90	0.91	0.87
	Multi-round SIFT + GNN	0.92	0.93	0.91
PreciseBugs	Pre-trained	0.22	0.16	0.35
	Label-only FT	0.31	0.43	0.25
	Single-round SIFT	0.33	0.46	0.25
	Multi-round SIFT	0.48	0.4	0.57
PreciseBugs Vuln. Type	*MSIVD*minus Input	0.46	0.49	0.44
	*MSIVD*minus Resource	0.58	**0.63**	0.51
	*MSIVD*minus Buffer	**0.59**	0.62	**0.57**

planation generation. Unlike prior ML-based vulnerability detection tools (e.g., Devign, IVDetect, LineVul, DeepDFA) that train solely on code or require extensive AST/CFG preprocessing, and unlike code-similarity methods (e.g., VUDDY, Vulpecker) and pattern-based methods (e.g., Chucky) that suffer from high false negatives or require manual pattern definition, MSIVD leverages rich vulnerability explanations available in curated datasets through self-instruct dialogue-based training.

Our key novelty lies in three contributions: (1) self-instruct dialogue augmentation that transforms classification datasets into multi-round teacher-student conversations about vulnerability characteristics; (2) multi-task learning that jointly optimizes for detection and explanation, improving generalization compared to single-task approaches; and (3) lightweight GNN adapters that incorporate dataflow analysis from program call graphs, enabling simultaneous transfer learning between LLM and GNN components. We evaluated MSIVD on BigVul and PreciseBugs datasets, achieving state-of-the-art results.

Critically, to address the risk of LLM data contamination identified in recent work, we collected a novel vulnerability dataset with samples exclusively filtered to vulnerabilities identified after our pretrained LLM's training cutoff. MSIVD outperforms prior work on this post-training-cutoff dataset, providing strong evidence that our approach generalizes beyond potentially contaminated training data. This work supports my thesis by demonstrating that program analysis complements LLMs through structured signals—specifically, dataflow properties via GNNs. By explicitly integrating these analysis-derived signals with LLM fine-tuning beyond autoregressive generation, MSIVD consistently outperforms pure learning-based or pure analysis-based

approaches for complex software maintenance tasks.

7 Node.js Vulnerability Detection with Program Analysis and pretrained LLMs

This chapter demonstrates how program analysis complements LLMs for vulnerability detection by providing dynamic dataflow properties as structured signals. While Chapter 6 used static dataflow analysis, this work leverages dynamic taint analysis to construct provenance graphs that capture the complete history of operations on tainted data during execution. These provenance graphs provide richer analysis signals that track not just what data flows where, but how it is transformed. By explicitly integrating these dynamic analysis-derived signals with LLMs beyond autoregressive generation, this chapter provides further evidence that analysis signals complement LLMs for software maintenance tasks.

As discussed in Chapter 6, we aim to improve the vulnerability detector from Chapter 3 for larger, real-world repositories. Chapter 6 extends *LLMAO* with *MSIVD*, which addresses the key limitation of context window sizing. *MSIVD* leverages the data flow analysis (DFA) of a program into trainable objectives for LLM-based vulnerability detection. DFA tracks how data moves through program variables by capturing control flow and data dependencies. However, not all information passed across a repository is represented by variables. A more specific static analysis technique is the creation of *provenance graphs*. Provenance graphs track data origins and transformations, and focuses on capturing relationships between data artifacts and processes. Provenance graphs investigate the "why" and "how" of data transformations, which are necessary for uncovering certain exploits. In this work, we further extend our previous vulnerability detector with provenance graphs, which are constructed through dynamic taint analysis that capture the complete history of operations applied to tainted values during a program's execution. Each node in a provenance graph represents a taint-related operation or value, while edges illustrate the flow of data between these nodes.

The popular Node.js ecosystem has become an attractive target for attackers. Two of the most serious vulnerabilities are Arbitrary Command Injection (ACI) and Arbitrary Code Execution (ACE), which allow attackers to execute malicious commands or code on the system that runs the application [22, 23]. Prior work such as NODEMEDIC-FINE used program synthesis to generate proof-of-concept exploits for Node.js packages for vulnerability confirmation. However, there are many instances where no exploit is produced, resulting in a high number of false positives that require manual analysis.

This work investigates whether ML-based approaches can predict ACE and ACI vulnerabilities in Node.js packages. We present a large-scale dataset of over 2,000 package-level vulnerabilities with human-reviewed labels. We conduct a comprehensive evaluation of approaches based

on large language models (LLMs), program analysis extended with ML approaches, and hybrid ML methods. We integrate the vulnerability detection engine of a prior program analysis tool, NODEMEDIC, with Graph Neural Networks, Random Forest, XGBoost, Logistic Regression, SVM, and LLMs to reduce false negatives.

Our results indicate that both traditional machine learning models and LLMs when combined with program analysis tools can predict Node.js package exploitability with high accuracy. Our work provides insights into the effectiveness and challenges of using LLMs for vulnerability detection in JavaScript packages.

7.1 Background and Related Work

7.1.1 Node.js Package Threat Model

Node.js is built on top of the V8 JavaScript engine. Node.js developers combine code into *packages*, which can import other packages as *dependencies* to use their public APIs (exported functions). Node.js provides powerful *sensitive APIs* [91, 92, 96, 97] that can dynamically generate code and execute shell commands.

In a real-world attack, a Node.js package that unsafely uses sensitive Node.js APIs is included as a *dependency* of a *victim* application. An attacker can be any user communicating with the victim application. Attacker-controlled input is passed from the victim application to the dependency's public API, which passes it to a sensitive API (e.g., `exec` [96]). We utilize an idealized model of the above scenario that echoes trust assumptions in prior work [13, 14, 113]. The attacker *directly* passes input to the dependency. We consider *all* public APIs of the dependency to be the attack surface of the package. We scope this work to focus on two types of severe attacks: arbitrary code execution (Arbitrary Code Execution (ACE)) [23] and arbitrary command injection (Arbitrary Command Injection (ACI)) [22]. An attacker capable of these can launch other attacks, e.g., directory traversal [21], by extension.

7.1.2 Dynamic Taint Analysis

Taint analysis specifies and checks policies governing *sensitive dataflow* with programs. A goal of the analysis is to detect flows from particular *sources*, e.g., API inputs, to *sinks*, e.g., functions with dangerous capabilities such as command execution. Taint analysis that runs during program execution is termed dynamic taint analysis. Dynamic taint analysis has been particularly efficacious for detecting code vulnerabilities of JavaScript (c.f. [3]).

Several tools perform taint analysis of Node.js packages [37, 50, 63, 89, 94, 100, 112]. We focus on leveraging NodeMedic-fine [13], which implements a taint *provenance analysis*, wherein a history of operations applied to tainted data is saved as *provenance graph* and can be used for further analysis. NodeMedic-fine uses the provenance graph for synthesizing proof of concept exploit to confirm the vulnerability. This work uses the provenance graph as input to a set of ML methods for vulnerability detection.

7.1.3 Provenance Graph

Provenance graphs generated by NODEMEDIC-FINE are data structures constructed through dynamic taint analysis that capture the complete history of operations applied to tainted values during a program's execution. Each node in a provenance graph represents a taint-related operation or value, while edges illustrate the flow of data between these nodes.

We reuse the toy package `toygrep` from NODEMEDIC [14] as an illustrative example to demonstrate how NODEMEDIC generates the provenance graphs using package source code and driver programs. This toy example (shown in Listing 7.1) exposes an API function called `grep`, which takes an argument `query` and executes the system command `grep [query]` without any sanitization.

The driver program (shown in Listing 7.2) simulates the behavior of an external input by creating a variable `x` marked as tainted using the `__set_taint__` function. Then this tainted variable is passed as the `query` argument to the `grep` function, triggering the execution of the system command.

```
1 function grep(query) {
2    exec("grep " + query);
3 }
```

Listing 7.1: Source code for the toy package `toygrep`

```
1 var PUT = require("toygrep");
2 var x = "tainted"; // {0:'0'}
3 __set_taint__(x);
4 try{PUT.grep(x);}
5 catch (e) {console.log(e)}
```

Listing 7.2: Driver program for the toy package `toygrep`

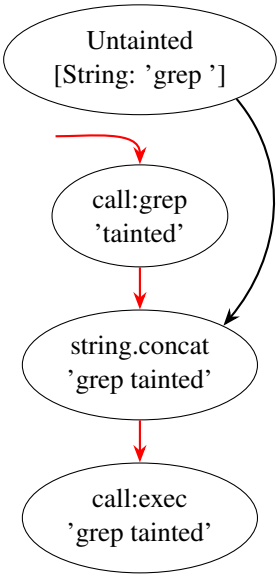

Figure 7.1: Provenance graph generated from the `toygrep` package. Upper-left section contains source code (Listing 7.1); Bottom-left section contains the driver program (Listing 7.2); Right section presents the provenance graph. A circle is a node. Black edges are untainted, red edges are tainted flows.

Figure 7.1 shows the provenance graph generated from the vulnerable toy package `grep` and its driver program. The graph illustrates how the input flows through the `grep` function call, is concatenated with the string `grep`, and ultimately reaches the sink, `exec`. Each node in a provenance graph captures key details about the operations and data flow within the program. Table 7.1 provides an overview of these attributes.

Table 7.1: Attributes of nodes in provenance graphs.

Attribute	Description
Operation	Type of operation, e.g., `call` or `Untainted`.
Value	Input provided to the operation.
File Path	File where the operation occurs.
Position	Start and end line/column in the file.
Tainted Status	Whether the data is tainted or untainted.
Flows From	Predecessor nodes in the data flow.
Sink Type	`spawn` or `exec`.

7.2 NODEMEDIC-LLM

In this work, we conduct a comprehensive evaluation of LLM-based methods, program analysis-based methods, ML-enhanced program analysis methods, and a hybrid approach that combines LLMs, NODEMEDIC-FINE, and GNNs for detecting vulnerabilities in JavaScript code. A summary of our approaches is shown in Figure 7.2. More concretely, we implement the following approaches with machine learning predictors that can help reduce the effort required for manual confirmation of package exploitability. (1) LLMs that do not rely on program analysis tools (Section 7.2.2). (2) NODEMEDIC-ML that applies various ML models (Random Forest, XGBoost, Logistic Regression, and SVM) to predict vulnerabilities based on the provenance graph output from the dynamic taint analysis tool NODEMEDIC-FINE (Section 7.2.3). (3)NODEMEDIC-GNN, which is setup similarly as NODEMEDIC-ML but uses Graph Neural Networks (Section 7.2.3). (4) NODEMEDIC-GNN-LLM, which combines the embedding layers of the GNN model and the pre-trained language models (LMs) (Section 7.2.4).

7.2.1 Dataset

The dataset used for our evaluation consists of 2,051 npm packages in which NODEMEDIC-FINE identified potential vulnerable calls to sinks. These packages were selected from a total of 33,011 npm packages analyzed for ACI and ACE vulnerabilities by NODEMEDIC-FINE and shared by the authors of NODEMEDIC-FINE.

Of the 2,051 potentially vulnerable npm packages, 1,465 have been confirmed to be vulnerable with working exploits. Of these, 728 were automatically confirmed by NODEMEDIC-FINE, while 737 were confirmed manually.

Since LLM methods, the LLM-GNN hybrid method, and ML-enhanced NODEMEDIC-FINE methods require parts of the data for training and validation, we randomly divided the 2,051 package dataset into three subsets (train, validation, and test) in an 8:1:1 ratio. The training set includes 1,640 packages, the validation set includes 205 packages, and the testing set includes 206 packages. For models requiring training, the training set is used to train the model, while

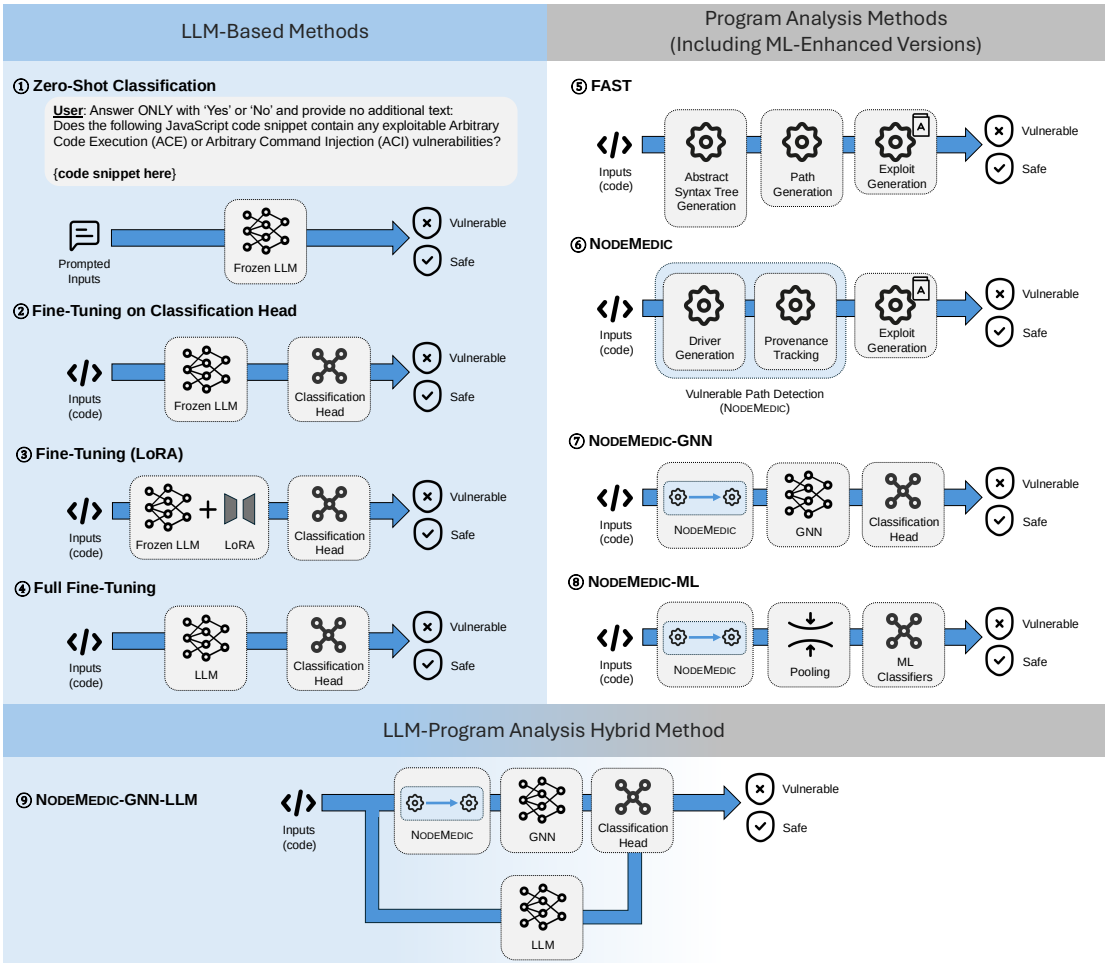

Figure 7.2: Nodemedic-LLM vulnerability detection approach

the validation set is used to select the best model model before evaluation. All models are then evaluated on the same testing set to ensure consistent performance reporting. Table 7.2 provides a detailed overview of these dataset splits.

7.2.2 LLM-based Methods

Common practices in leveraging large language models (LLMs) for vulnerability detection typically fall into two categories:

1. **Zero-shot and few-shot prompting**, where the model is provided with code snippets along with carefully designed natural language queries to identify security risks. This approach benefits from LLMs' generalization ability but often struggles with nuanced vulnerabilities that require deeper program understanding.
2. **Fine-tuning**, where the model is trained on labeled vulnerability datasets to learn domain-specific patterns. Fine-tuning can significantly improve detection accuracy but comes with high computational costs and data collection challenges. Other than full fine-tuning, there

Table 7.2: Overview of the dataset splits used in the evaluation. The table displays the total number of packages in each split (*train*, *validate*, and *test*), along with the number of vulnerable packages in each split, categorized into ACE and ACI. Numbers in parentheses indicate the count of vulnerable packages within each split.

Split	Total (Vuln)	ACE (Vuln)	ACI (Vuln)
train	1640 (1173)	335 (290)	1,305 (883)
validate	205 (140)	45 (38)	160 (102)
test	206 (152)	40 (32)	166 (120)
total	2,051 (1,465)	420 (360)	1,631 (1,105)

are also lightweight fine-tuning methods, such as LoRA fine-tuning [26, 44] and fine-tuning on only selected layers.

To systematically evaluate LLMs in vulnerability detection, we assess several models under different settings, including zero-shot classification, fine-tuning on a classification head, LoRA fine-tuning [26, 44], and full fine-tuning. We exclude few-shot learning in this work because the large input size of the code snippets makes it difficult to fit multiple samples into a reasonable context window. The code snippet given to LLMs is the file that contains potential sinks. If the file is too long to fit within the predefined context length, we truncate it, taking code immediately around the sink. This is based on our observations that the majority of vulnerable logic is local to the sink and the surrounding code in the dataset.

Zero-Shot Classification (① in Figure 7.2)

In this setting, we use an auto-regressive generation head that enables the LLM to generate a textual response indicating whether a given JavaScript package contains vulnerabilities. Zero-shot classification relies entirely on the LLM's pre-trained knowledge and ability to generate a relevant answer token by token in an auto-regressive manner.

For this evaluation, we prompt the LLM with a common query:

> **User:** Answer ONLY with 'Yes' or 'No' and provide no additional text:
> Does the following JavaScript code snippet contain any exploitable Arbitrary Code Execution (ACE) or Arbitrary Command Injection (ACI) vulnerabilities?
>
> `{code snippet here}`

For models that can run locally, we disable sampling during generation to ensure deterministic results. For some models that require a cloud-based API, the classification responses may vary slightly across multiple runs. Even though we instruct the model to output only "Yes" and "No," for models trained on a Chain-of-Thought (CoT) objective [129], the model may still produce extra text wrapped in special tokens as part of the reasoning process before the final answer. We ignore these additional tokens and only consider the final answer. The outputs are filtered based on the presence of "Yes" which is considered vulnerable, while all other cases, including those that generate neither "Yes" nor "No," are considered non-vulnerable.

Fine-Tuning on a Classification Head (② in Figure 7.2)

Instead of using a generation head, we attach a classification head on top of the base LLMs and fine-tune the model for the vulnerability detection task while keeping the base LLMs frozen (not involved in training). The classification head takes the embedding from the last non-padding position of the output from the last attention layer as input and produces an output shape of 2 logits, representing the two classes (vulnerable or non-vulnerable). We use the cross-entropy loss function to train the classification head with weights that correspond to the class imbalance in the dataset.

LoRA Fine-Tuning (③ in Figure 7.2)

LoRA (Low-Rank Adaptation) fine-tuning [26, 44] offers a lightweight approach to adapting LLMs without updating all parameters. Instead of modifying the entire model, LoRA injects low-rank adapters into selected layers of the model. These adapters are small, low-rank matrices that are learned during fine-tuning. Similar to the previous method, we attach a classification head on top of the base LLMs and fine-tune the model for the vulnerability detection task. However, in this case, we only update the low-rank adapters and the classification head, while the base LLMs remain frozen. The same cross-entropy loss function is used to train the classification head.

Full Fine-Tuning (④ in Figure 7.2)

In full fine-tuning, we update all parameters of the LLM and the classification head using our labeled vulnerability dataset, applying the same cross-entropy loss function as previously described.

During the fine-tuning of all the aforementioned LLM-based methods, all frozen parameters are stored in 4-bit NormalFloat (NF4) precision for memory efficiency [26], while the trainable parameters are in 16-bit BrainFloat (BF16) precision.

7.2.3 ML-enhanced Program Analysis Methods

NODEMEDIC-GNN and NODEMEDIC-ML utilize NODEMEDIC's taint provenance tracking component to create provenance graphs (as described in Section 7.1.3) in both the training and inference pipelines. The operation, tainted status, and sink type of each node are used as inputs. Additionally, the vulnerability type is included as an input for the entire graph. The 100 most common operations in our dataset (as described in Section 3.1) are assigned class numbers `0` to `99`. Class `100` is designated for less frequent operations, while class `101` is used for empty or missing operation attributes in the provenance graphs. Tainted statuses are encoded as class `0` for `False` (untainted), class `1` for `True` (tainted) and class `2` for missing attributes. Sink types are represented with class `0` for `spawn`, class `1` for `exec`, and class `2` for missing attributes. Vulnerability types are encoded as class `0` for ACE and class `1` for ACI vulnerabilities.

Each attribute is represented as a one-hot vector, where the corresponding class has a value of `1`, and all other classes have a value of `0`. The four one-hot vectors are then concatenated to form the embedding for a single node in the graph. Together, the graph's topology and the embeddings of its nodes make up the complete representation of the graph.

NODEMEDIC-GNN (⑦ in Figure 7.2)

The GNN component in NODEMEDIC-GNN starts with a Gated Graph Sequence Neural Network (GGNN) [71], which is a specialized type of neural network designed to learn from graph-structured data by capturing dependencies and relationships between nodes. The GGNN works by iteratively passing messages along edges, enabling each node to gather information from its neighbors and update its representation based on the graph's structure and features. In the final step, the learned abstract node embeddings are combined into a graph-level representation using Global Attention Pooling [71], resulting in the final graph embedding. The graph embedding is then fed into a classification head to predict vulnerability.

NODEMEDIC-ML (⑧ in Figure 7.2)

In NODEMEDIC-ML, only the node embeddings of the graph are taken into account, while the topology is disregarded. The embeddings of all nodes are first fed into a pooling layer to create a unified shape embedding vector that represents the entire graph, regardless of the number of nodes. The pooled embedding is then passed through machine learning classifiers to predict the vulnerability.

7.2.4 LLM-Program Analysis Hybrid Methods

To bridge the gap between program analysis and LLM-based reasoning, we evaluate a hybrid approach (⑨ in Figure 7.2) that combines NODEMEDIC-FINE's vulnerability path detection, graph neural networks (GNNs), and large language models (LLMs). This method, referred to as NodeMedic-GNN-LLM , integrates the strengths of both program analysis and LLMs to enhance vulnerability detection.

This hybrid approach uses the output from the GNN model and the output from the LLM as its embeddings. The two embeddings are concatenated and sent through a classification head to predict the vulnerability. The full model is trained end-to-end, with the GNN and LLM components updated simultaneously.

7.3 Experiment Setup

We first outline the model selection and implementation (Section 7.3.1) used in our evaluation. Next, we discuss the evaluation metrics (Section 7.3.2) and the system configuration (Section 7.3.3).

7.3.1 Model Selection and Implementation

We directly report the latest results from the NODEMEDIC-FINE experiments in Section 7.4. We evaluated FAST [1] against our dataset. FAST [48] uses constraint solving to generate *potential* exploits. Successful execution of the generated exploits could be used to confirm vulnerabilities,

[1]FAST implementation accessed at `https://github.com/fast-sp-2023/fast`

though FAST stops after exploit generation and thus can report false positives. For FAST, we enabled the `-X` flag to turn on auto-exploit generation. Additionally, for FAST, we used the `-t` flag to specify vulnerability types as `os_command` and `code_exec`, which correspond to ACI and ACE vulnerabilities, respectively.

For NODEMEDIC-ML, we evaluate logistic regression, support vector machine (SVM), random forest, and XGBoost. These methods are trained on provenance graphs generated by NODEMEDIC-FINE. For logistic regression, SVM, and random forest experiments, we used classes from the `scikit-learn` library, while the `xgboost` package was used for XGBoost experiments. For these machine learning baseline models, default hyperparameters were applied, and the random state was set to 42 for reproducibility.

For NODEMEDIC-GNN, we use the GatedGraphConv and GlobalAttentionPooling modules from the Deep Graph Library [126].

For the LLM-based methods, we experiment with DeepSeek-R1-Distill-Qwen-14B, DeepSeek-R1-Distill-Llama-8B, DeepSeek-R1-Distill-Qwen-7B [24], Llama-3.1-8B-Instruct [28], Qwen2.5-Coder-14B-Instruct, and Qwen2.5-Coder-7B-Instruct [144]. We utilized the implementation and parameters of the models provided by Hugging Face. Additionally, we experiment with two models evaluated via cloud APIs: OpenAI o3-mini-high and DeepSeek-R1 [24]. The random state is set to 0 to ensure reproducibility. For LoRA, we are using a rank of 256 and an alpha of 512 for all experiments. An ablation study on LoRA's hyperparameters is presented in Section 7.4.3.

7.3.2 Evaluation Metrics

NODEMEDIC-FINE, FAST, and zero-shot LLM models are evaluated directly on the test dataset, while other models that require training are trained on the training dataset and validated on the validation dataset during the training process. The best model from training is then evaluated on the test dataset. To assess the performance of each method, we employ the following metrics: $F1 = \frac{TP}{TP+0.5(FP+FN)}$, $Precision = \frac{TP}{(TP+FP)}$, $Recall = \frac{TP}{TP+FN}$ (where TN is true negative, TP is true positive, FP is false positive, and FN is false negative). These are the same metrics used for prior work on vulnerability detection [35, 68, 114, 142].

7.3.3 System Configuration

Evaluations that require only CPUs are conducted on a computing cluster. Each task runs individually in a virtually isolated environment with a 2-core CPU, which is part of an AMD EPYC 7742 processor, and 16 GB of RAM. The timeout for each task is set to 36 hours. For experiments that require GPUs, except for those where LLMs are undergoing full fine-tuning, each task is executed in a virtual environment with one NVIDIA H100 (80GB) GPU, two Intel Xeon 8480C PCIe Gen5 CPUs (each with 56 cores running at 2.0/3.8 GHz), and 2 TB of RAM. For LLMs that require full fine-tuning, we use a computing cluster with four NVIDIA H100 (80GB) GPUs, two Intel Xeon 8480C PCIe Gen5 CPUs (each with 56 cores running at 2.0/3.8 GHz), and 2 TB of RAM.

7.4 Results

Our experiments aim to address the following research questions:

RQ1: Effectiveness of LLMs. How effective are LLM-based methods in detecting exploitable vulnerabilities in Node.js packages compared to traditional program analysis approaches and ML-enhanced program analysis methods?

RQ2: Cross-Method Prediction Comparison. How do the predictions of graph-based models (e.g., GNN-enhanced NODEMEDIC-FINE) and LLM-based methods differ, and does their combination lead to improved vulnerability detection?

RQ3: Ablation Study on LLMs. How do different large language models and usage strategies (e.g., zero-shot, various fine-tuning methods) perform in our vulnerability detection task?

RQ4: Training and Inference Time. How do LLM-based methods compare to traditional program analysis approaches in terms of prediction latency and pre-prediction (training) time?

7.4.1 RQ1: Effectiveness of LLMs

To assess the effectiveness of LLM-based methods in detecting exploitable vulnerabilities, we compare them against traditional and ML-enhanced program analysis approaches. The evaluation focuses on F1 score, Precision, Recall, and Accuracy, where higher values indicate better detection performance.

Table 7.3 shows the full results of our evaluations. We include four naive baselines for comparison: Random ($P_{vuln} = 1/2$), Random ($P_{vuln} = 1173/1640$), Random ($P_{vuln} = 1$), and Random ($P_{vuln} = 0$). The first baseline randomly predicts vulnerabilities with a probability of $1/2$, the second baseline uses the ratio of vulnerable packages in the training split of our dataset, the third baseline always predicts vulnerabilities, and the fourth baseline never predicts vulnerabilities. The performance of methods using GNNs is based on the average of three runs, as the results are not deterministic when using GPUs. In contrast, other methods are deterministic and are evaluated with just one run.

Program Analysis Approaches. Traditional program analysis methods, such as FAST and NODEMEDIC-FINE, have high precision, but low recall, indicating that they often miss vulnerabilities. NODEMEDIC-FINE with auto-confirmation has better precision and recall than FAST, because FAST only generates potential exploits, some of which do not work.

ML-enhanced program analysis significantly improves performance: NODEMEDIC-GNN achieves the highest F1 score (0.943) among all methods. Other ML-enhanced methods (e.g., Random Forest, XGBoost, SVM) also perform well, with F1 scores ranging from 0.914 to 0.931. This indicates that combining machine learning with program analysis enhances detection accuracy and recall without significantly sacrificing precision.

LLM-based Methods. LLM-based methods exhibit significant variability based on whether they are used in a zero-shot setting or fine-tuned with various strategies. Overall, zero-shot LLMs tend to perform poorly. Even very large commercial models (e.g., OpenAI o3-mini-high) do not perform well in zero-shot settings. Fine-tuned LLMs generally achieve higher F1 scores. However, fine-tuning only the classification head results in notably lower performance compared to LoRA and full fine-tuning. We will provide a more detailed analysis of the results in Section

Table 7.3: F1 Score (F1), Precision (Prec), Recall (Rec), and Accuracy (Acc) for all program analysis-based methods and LLM-based methods. For LLMs, methods are categorized with the base LLMs. Higher metrics indicate better performance. - indicates that the calculation of such metrics is undefined due to division by zero. *Bold* indicates the best F1 score within each group; underline indicates the best F1 score overall.

Model	F1	Prec	Rec	Acc
Random ($P_{vuln} = 1/2$)	0.596	0.738	0.500	0.500
Random ($P_{vuln} = 1173/1640$)	0.727	0.738	0.715	0.602
Random ($P_{vuln} = 1$)	**0.849**	0.738	1.000	0.738
Random ($P_{vuln} = 0$)	-	-	0.000	0.262
FAST	0.699	0.915	0.566	0.641
NODEMEDIC-FINE				
–w/ auto confirmation	0.744	1.000	0.592	0.699
–w/ GNN	**0.943**	0.955	0.932	0.917
–w/ Random Forest	0.929	0.911	0.947	0.893
–w/ XGBoost	0.931	0.928	0.934	0.898
–w/ Logistic Regression	0.917	0.889	0.947	0.874
–w/ SVM	0.914	0.883	0.947	0.869
OpenAI o3-mini-high				
–zero-shot	0.733	0.978	0.586	0.684
DeepSeek-R1				
–zero-shot	0.858	0.842	0.875	0.786
Llama-3.1-8B-Instruct				
–zero-shot	0.844	0.741	0.980	0.733
–cls-head-only-ft	0.843	0.735	0.987	0.728
–lora-ft	0.922	0.876	0.974	0.879
–full-ft	0.849	0.738	1.000	0.738
–full-ft + GNN	0.850	0.761	0.967	0.748
Qwen2.5-Coder-7B-Instruct				
–zero-shot	0.353	0.943	0.217	0.413
–cls-head-only-ft	0.849	0.738	1.000	0.738
–lora-ft	0.900	0.857	0.947	0.845
–full-ft	**0.933**	0.907	0.961	0.898
–full-ft + GNN	0.919	0.888	0.952	0.875

7.4.3. Besides, adding NODEMEDIC-FINE features and GNN embeddings to LLMs does not always improve performance. For example, Qwen2.5-Coder-7B-Instruct with NODEMEDIC-FINE features and GNN embeddings performs worse than the same model without these features. We will discuss this in more detail in Section 7.4.2.

Overall, ML-enhanced program analysis methods (e.g., NODEMEDIC-GNN, F1: 0.943) outperform both traditional program analysis and even all fine-tuned LLMs. The best fine-tuned LLMs, Qwen2.5-Coder-7B (F1: 0.933), perform similarly to ML-enhanced methods but require significant training resources. Zero-shot LLMs are inconsistent and can sometimes perform worse than random baselines. Traditional program analysis methods struggle with recall,

which can limit their practical application in real-world scenarios. However, because they can generate proof-of-concept exploits, they may be more suitable for tasks where PoC or perfect precision is necessary. In contrast, ML-enhanced methods can offer a higher recall and higher F1 score solution. One advantage of LLM-based methods is that they do not require the same level of domain expertise in specific programming languages or security to design features or rules as program analysis methods do, making them easier to design and implement.

RQ1 Summary

GNN-enhanced NODEMEDIC-FINE performs best among all methods, while certain fine-tuned LLMs can achieve similar performance. Traditional program analysis methods struggle with low recall but can provide PoC exploits and perfect precision.

Table 7.4: Training and inference times for different models and methods. The inference time is measured per sample in a batched setting: calculated by dividing batch inference time by batch size for methods that support batch inference. Model saving and loading times are not included in the time measurement. *Other ML classifiers* refer to Random Forest, XGBoost, Logistic Regression, and SVM.

Model	Computation Time	
	Training	Inference
FAST	0min	31.6s
NODEMEDIC-FINE		
–w/ auto confirmation	0min	0.79s
–w/ GNN	25min	0.79s
–w/ other ML classifiers	22min	0.79s
DeepSeek-R1-Distill-Qwen-14B		
–zero-shot	0min	15.37s
Llama-3.1-8B-Instruct		
–zero-shot	0min	0.48s
–cls-head-only-ft	11mins	0.11s
–lora-ft	34mins	0.11s
–full-ft	24mins	0.10s
Qwen2.5-Coder-7B-Instruct		
–zero-shot	0min	0.25s
–cls-head-only-ft	10mins	0.10s
–lora-ft	30mins	0.10s
–full-ft	20mins	0.06s

7.4.2 RQ2: Cross-Method Prediction Comparison

Large Language Models

We examine how LLMs make predictions based on the input code. For this evaluation, we use DeepSeek-R1-Distill-Qwen-7B as a representative model. We compare the Shapley values of the classification-head-only fine-tuning version with the full fine-tuning to see how the model's focus shifts during training. Specifically, we choose the packages where, in the testing split, the classification-head-only fine-tuning makes incorrect predictions, while the fully fine-tuned version makes correct predictions. The classification head-only fine-tuning does not alter the LLM's core components, preserving most of its pre-trained parameter values. We use the implementation of SHAP (SHapley Additive exPlanations) [82] to generate the Shapley values.

We found that for non-vulnerable packages, the sink function *spawn* consistently exhibited a high Shapley value, strongly contributing to the model's classification as "not vulnerable."

Figure 7.3 is an example of an invulnerable package *third-party-resources-checker*, where the presence of *spawn* is the most significant feature in the prediction after full fine-tuning. This suggests that the presence of *spawn* is associated with safer execution patterns compared to other process creation functions.

One key reason for this is how *spawn* handles its arguments. Unlike *exec*, which interprets a string directly as shellcode, *spawn* takes a command and its arguments separately as elements of an array, and takes a second configuration object that only allows for shell meta-character evaluation if explicitly configured.

This design significantly reduces the risk of command injection attacks because an attacker must control both the command array and the configuration object (unless the package itself takes the unsafe action to allow shellcode evaluation). The need to control multiple arguments was specifically noted by NodeMedic-FINE as a challenge for confirming/dis-confirming exploitability of *spawn*.

For other cases, no specific pattern stands out, and most tokens contribute only marginally to the model's predictions. This suggests that, apart from certain key indicators like *spawn*, the vulnerability classification relies on a distributed set of features rather than any single dominant token. The Shapley values for these other tokens tend to be small and dispersed, indicating that their individual influence on the final prediction is limited. This behavior supports the idea that security vulnerabilities often result from complex interactions among different parts of the code, rather than being linked to the presence or absence of a single token.

Graph-Based Models

We compared the predictions of NODEMEDIC-GNN with the fully fine-tuned DeepSeek-R1-Distill-Qwen-7B. Among 206 packages in the test split, 178 packages were predicted correctly by the LLM while 189 packages were predicted correct by NODEMEDIC-GNN. Out of all the predictions on the test split made by LLM and NODEMEDIC-GNN: 171 cases were correctly classified by both classifiers. 10 cases were misclassified by both.

Additionally, we found that NODEMEDIC-GNN and LLMs perform very differently on ACI and ACE vulnerabilities. NODEMEDIC-GNN significantly outperforms LLMs on ACE vulnerabilities, achieving an F1 score of 0.955 compared to 0.889. In contrast, LLMs, along with

```
'use strict';

var fs = require('fs');
var phantomPath = require('phantomjs-prebuilt').path || '/usr/local/bin/phantomjs';
var Promise = require('promise');
var script = fs.realpathSync(__dirname + '/detect-phantom.js');
var spawn = require('child_process').spawn;

exports.check = function check (uri, whitelist) {
    return new Promise(function (resolve, reject) {
        var args = [script, uri];
        if (whitelist) args.push(whitelist);

        var phantomjs = spawn(phantomPath, args);
        var buffer = '';

        phantomjs.stdout.on('data', function(data) { buffer += data; });

        phantomjs.on('exit', function(code){
            var stdout = buffer.split("\n");
            stdout.pop();
            resolve([code, stdout]);
        });
    });
}
```

Figure 7.3: Shapley values of the full-fine-tuned DeepSeek-R1-Distill-Qwen-7B on a package *third-party-resources-checker*. The correct prediction is: "not vulnerable." Red values indicate a positive contribution to the vulnerable prediction, while blue values indicate a negative contribution. Darker colors represent a higher absolute Shapley value.

traditional program analysis methods (i.e., FAST and NODEMEDIC-FINE), perform better on ACI than on ACE. This indicates that the NODEMEDIC-GNN and LLMs may concentrate on different aspects of the package when making predictions. However, combining LLMs and GNNs does not always yield better results than using LLMs alone.

RQ2 Summary

LLMs' identification of non-vulnerable ACI relies on presence of the *spawn* sink, which is typically safer than *exec*. In other cases, no single token determines the prediction. Graph-based methods exhibit a different prediction distribution to LLMs across the test split, but combining LLMs and GNNs does not always improve performance.

7.4.3 RQ3: Ablation Study on LLMs

Our experiments show that the performance of LLMs in the vulnerability detection task is very sensitive to both the base model and the selected usage strategy (e.g., zero-shot, various fine-tuning methods). Table 7.3 includes key metrics for various LLM-based approaches under different operational configurations, including zero-shot and multiple fine-tuning strategies.

Zero-Shot In a zero-shot setting, OpenAI's o3-mini-high model achieves an F1 score of 0.733, which is lower than five out of the seven other models we evaluated for the zero-shot scenario. However, its metrics—specifically, high precision and moderate recall—are quite similar to those achieved by pure program analysis methods like NODEMEDIC-FINE. This suggests that the model tends to be conservative in its predictions.

Among other zero-shot LLMs, all models show fairly similar results, except for DeepSeek-R1-Distill-Qwen-7B and Qwen2.5-Coder-7B-Instruct, which have significantly lower F1 scores. DeepSeek-R1-Distill-Qwen-7B, in particular, sometimes outputs a final answer that contradicts its prior reasoning. The following are two examples where, during the reasoning, the model believes the code is exploitable, but the final answer is "no," and in the other example, it is the opposite.

Output 1:

$< think >$ ... I think **there are** exploitable ACE vulnerabilities here because the code allows for the execution of arbitrary functions based on the args parameter, which could be controlled by an attacker. $< /think >$ **No**

Output 2:

$< think >$ ... I think the code **does not have** any exploitable ACE or ACI vulnerabilities because the unshift method is overridden to do nothing, preventing any code execution. $< /think >$ **Yes**

LoRA vs. Full Fine-Tuning It is widely believed that a properly hyper-parameter-tuned Low-Rank Adaptation (LoRA) method can achieve performance similar to or even better than full fine-tuning [26, 44, 84]. We found similar results showing that LoRA fine-tuning performs almost as well as full fine-tuning for most models. For DeepSeek-R1-Distill-Llama-8B and Llama-3.1-8B-Instruct, LoRA fine-tuning even outperforms full fine-tuning. This suggests that LoRA fine-tuning can be a more efficient and effective approach for fine-tuning LLMs, especially when computational resources are limited. Figure 7.4 shows the comparison of various LoRA ranks on the models' F1 scores. A sudden increase in the F1 score is observed when the rank is increased, but additional increases in rank do not significantly enhance performance.

RQ3 Summary

LLM performance in vulnerability detection is highly sensitive to both model choice and usage strategy. In zero-shot settings, most models perform similarly. LoRA fine-tuning can achieve similar performance to full fine-tuning, and in some cases, it can even outperform full fine-tuning.

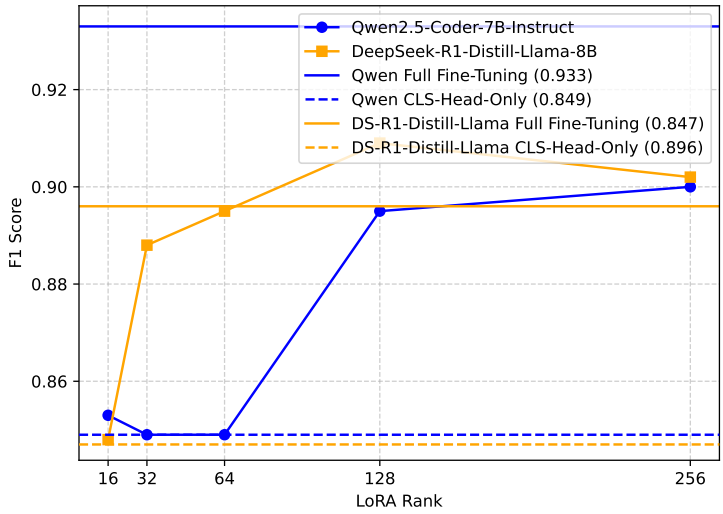

Figure 7.4: Comparison of various LoRA ranks on the F1 score of DeepSeek-R1-Distill-Llama-8B and Qwen2.5-Coder-7B-Instruct. The F1 score is computed based on the test split. Full fine-tuning and classification-head-only fine-tuning are included for comparison.

7.4.4 RQ4: Training and Inference Time

To compare the computational efficiency of LLM-based methods against traditional and ML-enhanced program analysis approaches, we analyze both training time (pre-prediction) and inference time (prediction latency), as presented in Table 7.4. All full fine-tuning of the LLMs is performed on four (4) GPUs, unlike other methods for LLMs where experiments are conducted using one (1) GPU. For FAST, the inference time is the median overhead reported in the paper [48] that introduces FAST. The graph generation times of NODEMEDIC-FINE represent the average tool runtime reported in the paper [14], which is 0.79 seconds per output. Some methods of DeepSeek-R1-Distill models are not reported because they share the same model structure as the non-distilled models, and their performance is expected to be similar.

Full fine-tuning or LoRA fine-tuning of LLMs usually takes more training time than ML-enhanced program analysis tools. However, the inference time for fine-tuned LLMs is quicker than that of both program analysis methods and ML-enhanced program analysis methods when processing in batches. FAST and NODEMEDIC-FINE provide a PoC exploit, while the other methods cannot.

RQ4 Summary

LLM-based methods have slightly higher training costs compared to ML-enhanced program analysis methods, but they remain within a reasonable range for practical use. Regarding inference time, fine-tuned LLMs are significantly faster (up to 10 times) than both program analysis methods and ML-enhanced program analysis methods, although the cost of hardware resources should be taken into account.

7.5 Conclusion

This chapter presented NodeMedic-LLM, the first comprehensive evaluation of hybrid approaches combining dynamic taint analysis with machine learning for Node.js vulnerability detection. Unlike prior work (e.g., NodeMedic-fine) that relies solely on program synthesis to generate proof-of-concept exploits (resulting in high false positives requiring manual analysis), and unlike pure static analysis tools for Node.js (e.g., SYNODE, Nodest, AFFOGATO) that lack the semantic understanding to reduce false negatives, our hybrid approach integrates provenance graphs from dynamic taint analysis with multiple ML techniques.

We evaluated four distinct approaches on a large-scale dataset of 2,051 npm packages with human-reviewed labels for ACE and ACI vulnerabilities: (1) pure LLM-based methods without program analysis; (2) NodeMedic-ML using traditional ML models (Random Forest, XGBoost, Logistic Regression, SVM) on provenance graphs; (3) NodeMedic-GNN using graph neural networks on provenance graphs; and (4) NodeMedic-GNN-LLM combining GNN and LLM embeddings. Our results demonstrate that NodeMedic-GNN achieves an F1 score of 0.943, significantly outperforming both pure program analysis approaches and pure LLM methods.

This work supports my thesis by demonstrating that program analysis complements LLMs through structured signals—specifically, dynamic dataflow properties from taint provenance graphs. By explicitly integrating these analysis-derived signals with LLMs and GNNs beyond autoregressive generation, NodeMedic-LLM consistently outperforms pure learning-based or pure analysis-based approaches. The quantitative evidence (F1 score of 0.943) shows that integrating dynamic analysis signals that track data transformations during execution with LLM representations improves software maintenance tasks.

8 Conclusion

This dissertation has explored the integration of Large Language Models (LLMs) with traditional program analysis techniques to address critical challenges in software evolution. Through a series of novel approaches and empirical evaluations, I have demonstrated that combining the strengths of LLMs with established software engineering methodologies yields superior results compared to either approach in isolation. My work spans multiple dimensions of software evolution, including fault localization, automated program repair, program transpilation, and security vulnerability detection.

8.1 Summary of Contributions

8.1.1 Bidirectional Fine-tuning for Fault Localization

I developed a bidirectional fine-tuning technique that enables LLMs to perform effective fault localization without relying on pre-written tests (Chapter 3) [139]. By adapting traditionally left-to-right LLMs to understand code in a bidirectional manner, I demonstrated significant improvements in identifying faulty lines of code. This approach not only assists in debugging but also proves effective in detecting runtime security vulnerabilities, addressing a critical gap in existing fault localization techniques.

8.1.2 LLM Entropy for Automated Program Repair

I introduced a novel application of LLM entropy values to enhance automated program repair (Chapter 4) [143]. My research showed that LLM entropy—a measure of model uncertainty—can be leveraged to improve all three critical stages of program repair: fault localization, patch testing efficiency, and plausible patch ranking. The empirical results demonstrated that this hybrid approach outperforms both traditional APR techniques and pure LLM-based methods, achieving an 18% higher precision in classifying plausible patches while remaining effective for patches produced by modern ML-based tools.

8.1.3 Verified Rust Transpilation

VERT, my verified equivalent Rust transpilation framework, combines LLMs with verification harnesses to ensure functional equivalence between source and target code (Chapter 5) [141]. This approach addresses the hallucination problem inherent in LLMs by incorporating formal

verification techniques, resulting in transpiled code that is not only functionally correct but also more idiomatic than code produced by traditional transpilers. The evaluation on real-world repositories confirms that this approach strikes an optimal balance between correctness guarantees and code naturalness.

8.1.4 Multi-task Security Vulnerability Detection

Building upon the fault localization work, I extended the approach to security vulnerability detection through multi-task instruction-tuning of LLMs (Chapter 6) [142]. By training models to simultaneously identify vulnerable code and explain exploitation vectors, I created a more comprehensive vulnerability detection system. The experiments demonstrate that this approach effectively detects vulnerabilities spanning multiple files across large repositories, addressing the limitations of line-level fault localization.

8.1.5 Hybrid Node.js Vulnerability Detection

NodeMedic represents a hybrid approach that combines program analysis with LLMs for detecting exploitable vulnerabilities in Node.js packages (Chapter 7). By integrating taint provenance tracking with graph neural networks and LLMs, I achieved superior detection performance compared to existing methods. The comprehensive evaluation shows that NodeMedic-GNN achieves an F1 score of 0.943, significantly outperforming traditional program analysis approaches and demonstrating the value of combining multiple techniques for vulnerability detection.

8.2 Key Insights and Implications

Several important insights emerge from this research that have broader implications for the field of software engineering.

While LLMs excel at generating natural, human-like code, they fundamentally operate as black-box models with inherent limitations in understanding program semantics and correctness. My work demonstrates that these limitations can be mitigated by integrating LLMs with traditional program analysis techniques, creating systems that leverage the strengths of both approaches.

The entropy values produced by LLMs during generation provide valuable signals about model uncertainty, which can be harnessed for tasks beyond code generation. This research shows that these entropy values effectively identify potential fault locations and help prioritize candidate patches, offering a novel perspective on how LLM internals can inform software engineering tools.

Verification of LLM outputs is essential for critical software engineering tasks. The transpilation work demonstrates that combining LLM generation capabilities with formal verification techniques produces results that are both correct and natural, addressing a fundamental tension in automated code transformation.

Multi-task learning enables LLMs to develop more nuanced understanding of software vulnerabilities by simultaneously learning to identify vulnerable code and explain exploitation vec-

tors. This approach produces models that can reason about security implications across file boundaries, addressing limitations of traditional vulnerability detection methods.

Most importantly, this research confirms that hybrid approaches combining neural and symbolic methods consistently outperform pure neural or pure symbolic approaches across various software evolution tasks. This finding has significant implications for the design of future software engineering tools, suggesting that the most effective systems will integrate multiple paradigms rather than relying exclusively on either traditional program analysis or deep learning.

8.3 Limitations and Future Work

Despite the advances presented in this dissertation, several limitations remain to be addressed in future work, while recent developments in LLMs present both new opportunities and continued challenges.

The scalability of these approaches to very large codebases remains a challenge, particularly for tasks requiring whole-program analysis. However, recent advances in LLM context windows (with models now supporting millions of tokens) present new opportunities for applying these techniques to larger codebases without the segmentation strategies employed in this work. Future research should explore how these expanded context windows can be leveraged while maintaining the benefits of the hybrid approaches developed here.

While the transpilation work focused on Rust as a target language, extending these techniques to other language pairs would provide valuable insights into the generalizability of the approach. Additionally, exploring bidirectional transpilation (i.e., converting code back to its original language) could serve as an additional verification mechanism. The emergence of agentic LLMs that can autonomously call verification tools presents exciting possibilities for self-checking transpilation systems that could iteratively improve their outputs.

The security vulnerability detection work currently focuses on specific vulnerability types in Node.js applications. Expanding this approach to cover a broader range of vulnerability classes and programming languages would increase its practical utility. Modern LLMs demonstrate improved zero-shot performance on fault localization tasks, potentially reducing the need for extensive fine-tuning. However, the specialized knowledge captured through the fine-tuning approaches in this dissertation (particularly the multi-task learning for vulnerability explanation) remains valuable for achieving high precision in security-critical applications.

The interpretability of hybrid neural-symbolic systems remains limited, making it difficult for developers to understand why certain code is flagged as vulnerable or why specific repairs are suggested. Future work should explore techniques for making these systems more transparent and explainable, potentially leveraging the improved reasoning capabilities of newer LLMs to provide better explanations of their decisions.

Despite rapid advances in LLM capabilities, there remains significant value in the program analysis integration and fine-tuning optimization techniques presented in this work. Models continue to be expensive to pre-train, and the techniques developed in this thesis can be valuable for generating higher quality training data without dependency on human labelers. The entropy-based approaches for patch ranking and the multi-task learning frameworks can serve as

automated data curation mechanisms, identifying high-quality code examples and vulnerability patterns for training future models.

Reinforcement learning presents an underexplored opportunity, particularly for security-related data flows. The provenance graphs and taint analysis techniques developed in this work could serve as reward signals for reinforcement learning systems, enabling models to learn optimal security practices through interaction with program analysis tools. This could lead to LLMs that not only detect vulnerabilities but also learn to write more secure code through reinforcement.

The parameter-efficient fine-tuning techniques (e.g., LoRA) demonstrated in this work remain relevant even as base models grow larger. These approaches enable rapid adaptation of large models to specific domains and tasks without the computational overhead of full fine-tuning, making specialized software engineering applications more accessible to researchers and practitioners with limited computational resources.

Future work should also investigate how the hybrid approaches developed here can be adapted to leverage emerging LLM capabilities while maintaining their core advantages. For instance, combining the bidirectional fine-tuning techniques with modern instruction-following models could yield even more effective fault localization systems, while the verification frameworks developed for transpilation could be extended to work with agentic systems that can autonomously invoke multiple verification tools.

8.4 Concluding Remarks

This dissertation has demonstrated that software engineering is indeed an evolving process that benefits from a holistic understanding of software properties. By addressing the limitations of LLMs through integration with program analysis techniques, I have created effective tools for fault localization, automated program repair, program transpilation, and security vulnerability detection.

The work contributes to a growing body of evidence suggesting that the future of software engineering tools lies not in replacing traditional techniques with deep learning, but in thoughtfully combining them to create systems that leverage the strengths of both approaches. As software systems continue to grow in complexity and importance, such hybrid approaches will be essential for maintaining software quality, security, and evolvability.

The tools and techniques presented in this dissertation are open-source and available to the research community, providing a foundation for future work in this rapidly evolving field. I hope that these contributions will inspire further research into the integration of LLMs with traditional software engineering methodologies, ultimately leading to more powerful and practical tools for software evolution.

Bibliography

[1] 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. 3.1, 3.3.2

[2] 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, 3.1

[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, 2017. URL `https://doi.org/10.1145/3106739`. 7.1.2

[4] 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`. 5.1.1

[5] 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

[6] 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. 3.2.2

[7] 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. 5.3.2

[8] 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, 2.1.2

[9] 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, 3.2.1

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

sandboxing using *WebAssembly*. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1975–1992, 2022. 5.1.2, 5.2.2

[11] 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

[12] 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

[13] Darion Cassel, Nuno Sabino, Ruben Martins, and Limin Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node. js vulnerabilities. 7.1.1, 7.1.2

[14] Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node. js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1101–1127. IEEE, 2023. 7.1.1, 7.1.3, 7.4.4

[15] 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, 2.1.2

[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, 3, 3.2.1, 3.2.1

[17] 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`. 5.1.1

[18] 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`. 5.1.1

[19] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 2001. 2.2.3, 5.2.5

[20] 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`. 2.2.3, 5.2.5

[21] The MITRE Corporation. CWE-22: Improper Limitation of a Pathname to a Re-

stricted Directory ('Path Traversal'), 2020–. https://cwe.mitre.org/data/definitions/22.html. 7.1.1

[22] The MITRE Corporation. CWE - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection') (4.3), 2020–. https://cwe.mitre.org/data/definitions/77.html. 7, 7.1.1

[23] The MITRE Corporation. CWE - CWE-94: Improper Control of Generation of Code ('Code Injection') (4.3), 2020–. https://cwe.mitre.org/data/definitions/94.html. 7, 7.1.1

[24] Daya Guo DeepSeek-AI, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. 7.3.1

[25] 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. 2.1.2

[26] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems*, 36, 2024. 2, 7.2.2, 7.2.2, 7.2.2, 7.4.3

[27] 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. 4.1

[28] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. 7.3.1

[29] 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. 5.1.2, 5.3.2

[30] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. A C/C++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020. 6.3.1, 6.3.1

[31] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. 6.3.2

[32] 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. 2.2.3, 5.2.5

[33] 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. 5.3.1

[34] 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. 2.1.2, 4.3.1

[35] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022. 6.3.1, 6.3.1, 6.3.2, 7.3.2

[36] 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

[37] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: Runtime detection of injection attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018. 7.1.2

[38] 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. 4, 4.1, 4.3.2, 4.3.4

[39] Ye He, Zimin Chen, and Claire Le Goues. PreciseBugCollector: Extensible, executable and precise bug-fix collection: Solution for challenge 8: Automating precise data collection for code snippets with bugs, fixes, locations, and types. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1899–1910. IEEE, 2023. 6.3.1

[40] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022. 6.1, 6.3.1, 5

[41] 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. 2.1.2, 4

[42] 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. 4.1

[43] 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. 3.3.6

[44] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 2, 7.2.2, 7.2.2, 7.4.3

[45] 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. 5.3.1

[46] 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

[47] 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, 3.3.1, 4.3.2

[48] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, VN Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1059–1076. IEEE, 2023. 7.3.1, 7.4.4

[49] 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

[50] R. Karim, F. Tip, A. Sochurkova, and K. Sen. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering*, 2018. 7.1.2

[51] Misoo Kim, Youngkyoung Kim, Kicheol Kim, and Eunseok Lee. Multi-objective optimization-based bug-fixing template mining for automated program repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022. 4

[52] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE symposium on security and privacy (SP)*, pages 595–614. IEEE, 2017. 6.1, 5

[53] 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. 5.4

[54] 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. 3.3.4

[55] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022. 6.3.6

[56] 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. 5.1.1

[57] 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

[58] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014. 4.1

[59] 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. 4.1

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

[61] 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. 4.3.1

[62] 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. 5.3.1, 5.3.2

[63] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. *Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis*. 2021. 7.1.2

[64] 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. 3.1

[65] 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, 3.3.2, 3.3.3, 3.3.4, 3.3.7

[66] 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`. 3.3.1

[67] 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. 3.3.2, 3.3.3

[68] Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021. 6.1, 6.3.1, 6.3.2, 6.2, 7.3.2

[69] 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. 3.1, 4.2.2

[70] 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, 3.3.4

[71] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015. 7.2.3

[72] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an au-

tomated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pages 201–213, 2016. 6.1

[73] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018. 6.3.2

[74] 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. 5.1.2

[75] 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. 5.3.1, 6.2

[76] 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, 4.2.2

[77] 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, 4.3.3, 4.3.4

[78] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. 6.3.2

[79] 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

[80] 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. 3.1

[81] 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, 3.3.1, 3.3.3, 3.3.4, 3.3.7

[82] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and

R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf. 7.4.2

[83] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi: 10.1109/TSE.2019.2946563. 6.1

[84] Yuren Mao, Yuhang Ge, Yijiang Fan, Wenyi Xu, Yu Mi, Zhonghao Hu, and Yunjun Gao. A survey on lora of large language models. *Frontiers of Computer Science*, 19(7):197605, 2025. 7.4.3

[85] 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

[86] 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. 3.1, 4.2.2

[87] 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. 3.3.1, 3.3.2, 3.3.3

[88] 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

[89] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association. 7.1.2

[90] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. {VulChecker}: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6557–6574, 2023. 5

[91] Mozilla. JavaScript eval function documentation, 2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval. 7.1.1

[92] Mozilla. JavaScript Function constructor documentation, 2022. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function. 7.1.1

[93] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Com-*

puter and communications security, pages 529–540, 2007. 6.1

[94] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: Feedback-driven static analysis of Node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019. 7.1.2

[95] 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, 3.2.1

[96] Node.js. Node.js exec API documentation, 2022. `https://nodejs.org/api/child_process.html#child_processexeccommand-options-callback`. 7.1.1

[97] Node.js. Node.js execSync API documentation, 2022. `https://nodejs.org/api/child_process.html#child_processexecsynccommand-options`. 7.1.1

[98] 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

[99] 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. 3.3.4

[100] Nishant Patnaik and Sarathi Sahoo. Javascript static security analysis made easy with JSPrime. In *Blackhat USA*, 2013. 7.1.2

[101] 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. 4.1

[102] 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. 4.1

[103] 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. 2.1.2, 4

[104] 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. 3.3.1

[105] 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. 2.1.2

[106] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Pro-*

cessing Systems, 33, 2020. 1, 5.3.3

[107] 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. 5.3.2, 6.2.2, 6.3.2

[108] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018. 6.3.2

[109] 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. 3.1

[110] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3), jun 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187673. URL `https://doi.org/10.1145/2187671.2187673`. 6.1

[111] 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`. 4, 4.1, 4.4

[112] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting Taint Specifications for JavaScript Libraries. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020. 7.1.2

[113] Cristian-Alexandru Staicu, M. Pradel, and B. Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *NDSS*, 2018. 7.1.1

[114] Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. 6.1, 6.3.1, 6.3.2, 6.2, 7.3.2

[115] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*, 2022. 5.3.2, 5.3.3

[116] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13. 6.1

[117] 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. 4, 4.1, 4.3.2, 4.3.4

[118] 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. 2.1.2, 4.3.1

[119] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. *Natural language processing with transformers*. O'Reilly Media, Inc., 2022. 3, 3.2.1

[120] Unknown. C2rust. `https://c2rust.com/`, 2024. 5.1.2, 5.3.2

[121] Unknown. bindgen. `https://github.com/rust-lang/rust-bindgen`, 2024. 5.1.2

[122] Unknown. citrus. `https://gitlab.com/citrus-rs/citrus`, 2024. 5.1.2

[123] Unknown. claude. `https://www.anthropic.com/index/introducing-claude`, 2024. 5.3.2

[124] 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. 5.2.5

[125] 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. 2.1.1, 3.2.1, 3.2.2

[126] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. 7.3.1

[127] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, 2022. 6.3

[128] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021. 6.3.2

[129] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. 7.2.2

[130] 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. 4.1

[131] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance 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. 3.3.1, 3.3.7

[132] Robert F Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages

1–3, 2007. 3.3.7, 3.3.7

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

[134] 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

[135] 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. 4, 4.2.2

[136] 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, 4, 4.1, 4.3.4

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

[138] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013. 6.1

[139] 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. 1, 1.2.1, 6.2, 8.1.1

[140] 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. 5.3.1

[141] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. Vert: Verified equivalent rust transpilation with few-shot learning. *arXiv preprint arXiv:2404.18852*, 2024. 1.2.3, 8.1.3

[142] Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024. 1.2.4, 7.3.2, 8.1.4

[143] Aidan ZH Yang, Sophia Kolak, Vincent Hellendoorn, Ruben Martins, and Claire Le Goues. Revisiting unnaturalness for automated program repair in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 626–626. IEEE Computer Society, 2025. 1, 1.2.2, 8.1.2

[144] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su,

Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024. 7.3.1

[145] 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. 4, 4.1, 4.4

[146] 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. 4.1

[147] Imam Nur Bani Yusuf and Lingxiao Jiang. Your instructions are not always helpful: Assessing the efficacy of instruction fine-tuning for software vulnerability detection. *arXiv preprint arXiv:2401.07466*, 2024. 6.3.1, 6.3.6

[148] 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. 5.1.2, 5.3.2

[149] 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. 3.3.1, 3.3.7, 6.1, 6.3.1, 6.3.1