Large Language Models for Test-Free Fault Localization

Aidan Z.H. Yang
aidan@cmu.edu
Carnegie Mellon University
Pittsburgh, United States

Claire Le Goues
clegoues@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, United States

Ruben Martins
rubenm@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, United States

Vincent J. Hellendoorn
vhellendoorn@cmu.edu
Carnegie Mellon University
Pittsburgh, United States

ABSTRACT

Fault Localization (FL) aims to automatically localize buggy lines of code, a key first step in many manual and automatic debugging tasks. Previous FL techniques assume the provision of input tests, and often require extensive program analysis, program instrumentation, or data preprocessing. Prior work on deep learning for APR struggles to learn from small datasets and produces limited results on real-world programs. Inspired by the ability of large language models (LLMs) of code to adapt to new tasks based on very few examples, we investigate the applicability of LLMs to line level fault localization. Specifically, we propose to overcome the left-to-right nature of LLMs by fine-tuning a small set of bidirectional *adapter* layers on top of the representations learned by LLMs to produce *LLMAO*, the first language model based fault localization approach that locates buggy lines of code without any test coverage information. We fine-tune LLMs with 350 million, 6 billion, and 16 billion parameters on small, manually curated corpora of buggy programs such as the *Defects4J* corpus. We observe that our technique achieves substantially more confidence in fault localization when built on the larger models, with bug localization performance scaling consistently with the LLM size. Our empirical evaluation shows that *LLMAO* improves the Top-1 results over the state-of-the-art machine learning fault localization (MLFL) baselines by 2.3%-54.4%, and Top-5 results by 14.4%-35.6%. *LLMAO* is also the first FL technique trained using a language model architecture that can detect security vulnerabilities down to the code line level.

CCS CONCEPTS

• **Software and its engineering** → **Software functional properties**; • **Computing methodologies** → **Neural networks**.

ACM Reference Format:
Aidan Z.H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623342

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623342

1 INTRODUCTION

Fault localization (FL) [1–4] 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. *Spectrum based fault localization (SBFL)* approaches, such as Tarantula [5] or Ochiai [6], apply statistical analysis on the coverage data of failed/passed tests to compute the suspiciousness of code elements. SBFL relies exclusively on test coverage and is thus less applicable for data-driven defects; it is also sensitive to properties of the underlying test suite like coverage and numbers of passing and failing tests [1]. *Mutation based fault localization (MBFL)* (like FIFL [7] or Metallaxis [8]) also analyzes test case behavior to localize faults, but uses mutation analysis to assess the concrete impact of particular code lines on test outcomes. While effective, MBFL approaches are computationally intensive and their performance is highly variable [9].

Recent advances in *Machine learning based fault localization (MLFL)*, like DeepFL [2], DEEPRL4FL [10], and GRACE [3]

use machine learning to relate code, test, or execution features to the likelihood of faultiness for a program entity. MLFL techniques learn to detect faulty lines of code from information including suspiciousness scores from existing SBFL and MBFL techniques (e.g., TRANSFERFL [11]), fault-proneness features like code complexity metrics (e.g., DeepFL [2]), or the test coverage matrix (DEEPRL4FL [10]), among others. These approaches speak to the potential that increasingly powerful machine learning models have for supporting debugging tasks.

Indeed, Deep learning (DL) has shown promise for many code related tasks, such as program synthesis [12, 13]. The most effective DL models for both natural language and code related tasks are large language models (LLMs), such as Codex [14] and GPT-4 [15]. This class of models trains many billions of parameters with even more tokens of training data, which tends to yield highly flexible and powerful text generators. LLMs' utility for code generation and the fact that they are trained on an abundance of publicly-available code [14] 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 [16], which is itself a function of the model and training data size.

For instance, LLM performance on program synthesis benchmarks increases linearly with the magnitude (log scale) of the number of parameters in the model [17]. This suggests that there is substantial performance to be unlocked for software engineering tasks by leveraging the largest publicly available language models. However, most existing work in this space to date either trains small models from scratch [2, 10, 18], or fine-tunes modest-sized models [11], missing out on the scale of state-of-the-art LLMs.

This is in part because 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 [13, 14, 19, 20] are trained to generate code in a left-to-right manner, with each token predicted from its preceding context. We posit that 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.

In this paper, 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* [21], 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. We 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. Our approach does not need to run or analyze the test cases or examine the program behavior on test cases to perform localization. Because the approach is lightweight, it can effectively fine-tune existing LLMs for particular languages (we show applicability to C, Java, and Python), or particular defect classes (we show applicability to functional defects and security vulnerabilities), with a relatively small amount of training data.

In summary, we make the following contributions.

- ***LLMAO.*** We propose a technique that uses different configurations of language models to predict faulty lines across three languages and two different application domains.
- **Novel large language model based learning for FL**. We showed that with fine-tuning on top of off-the-shelf large language models, we can achieve a higher fault detection rate than previous MLFL techniques without the use of test cases.
- **DL based security vulnerability detection**. *LLMAO* is the first MLFL technique that can detect code line level vulnerabilities in the security domain.
- **Empirical evaluation**. We evaluated *LLMAO* against recent state-of-the-art FL models to show its effectiveness in fault localization.

```
1  public StrBuilder appendFixedWidthPadRight(Object, int, char) {
2    ...
3    if (width > 0) {
4      ensureCapacity(size + width); // SBFL=0.35
5      String str = (obj == null ? getNullText()
6          : obj.toString()); //SBFL=0.35
7      int strLen = str.length(); //SBFL=0.35
8      ...
9  public StrBuilder appendFixedWidthPadLeft(Object, int, char) {
10  // relevant code identical to the above
11     ...
12 public String getNullText(){
13   return nullText;  // SBFL=0.71
14 }
```

(a) Code snippet implicated in Apache Commons Lang bug #47 from *Defects4J*. Both methods throw a `NullPointerException` when `getNullText()` also returns `null` (line 7). The developer addressed this by adding null checks after the assignment to `str` (not shown). Select lines are annotated with Ochiai [6] suspiciousness score.

```
1  public void testLang412Right() {
2      StrBuilder sb = new StrBuilder();
3      sb.appendFixedWidthPadRight(null, 10, '*');
4      assertEquals( "Failed to invoke appendFixedWidthPadRight",
5      "**********", sb.toString());
6  } //Test fails due to NullPointerException in appendFixedWidthPadRight()
7  public void testLang412Left() {
8      StrBuilder sb = new StrBuilder();
9      sb.appendFixedWidthPadLeft(null, 10, '*');
10     assertEquals( "Failed to invoke appendFixedWidthPadLeft",
11     "**********", sb.toString());
12 } //Test fails due to NullPointerException in appendFixedWidthPadLeft()
```

(b) Lang's bug #47 and corresponding failed tests

Figure 1: Apache Commons Lang Bug #47, from *Defects4J*

- **Artifact availability**. Our data, tool, and model checkpoints are available at Figshare.[1]

2 MOTIVATION

In this section, we discuss in detail two real-world bugs that test-based FL techniques struggle to clearly localize. We use these examples to motivate why we propose a novel language model based fault localization technique that shifts the dependence on tests to an LLM's latent understanding of source code.

2.1 General Logic Defects

Consider Figure 1a, which shows snippets of code from two methods in the Apache Commons Lang project. Lang-47 (i.e., bug #47 of the Lang project) from the *Defects4J* (V1.2.0) [21] dataset highlights a null pointer exception that can be triggered in both of these methods, for the same reason. The issue was addressed by adding the null pointer check and initialization shown starting on line 7 in `appendFixedWidthPadRight`; the identical code and block added in `appendFixedWidthPadLeft` is elided for brevity.

[1] https://figshare.com/s/35f36bff735d3c805a89

Given tests, we can use the Ochiai SBFL formula [6] to calculate code line suspiciousness scores to help pinpoint this bug. SBFL techniques in general compute suspiciousness by applying a formula to each entity (line, in this case) in the codebase based on test coverage information for passing and failing tests. Specifically, Ochiai counts for each code line (ℓ) the number of failed tests covering ℓ (ℓ_f) or not covering ℓ (n_f), and the number of passed tests covering ℓ (ℓ_p) or not covering ℓ (n_p). The suspiciousness score of a code line (ℓ) is then $Ochiai(\ell) = \ell_f(\ell_f + \ell_p)^{(-\frac{1}{2})}(\ell_f + n_f)^{(-\frac{1}{2})}$.

Several tests in the Apache Commons Lang test suite execute this code. The two that throw null pointer exceptions, demonstrating the bug, are shown in Figure 1b. Five others (not shown) execute these two methods as well, but are passing.

Figure 1a shows Ochiai scores computed using these tests. The scores demonstrate a common limitation of SBFL, which is that it cannot disambiguate between lines in a single straight-line block, as shown in `appendFixedWidthPadRight`. `testLang412Right()` executes lines 1–7, corresponding to the `then` block of the check on line 3. This computation is also misled by the small number of triggering tests: both failing tests cover `getNullText`, while only two of the five passing tests do. Line 13 in Figure 1a has a much higher score than the code in the two methods that call it.[2]

MLFL techniques like DeepFL [2] use other features on top of SBFL suspiciousness scores for training data, like textual similarity information, to guide their model to detect faulty methods. However, DeepFL only has confidence in method-level fault localization, with limited results at the statement level.

Our technique can detect line 7 from Figure 1a as highly suspicious. It assigned a score of 0.33 on line 7, and ranks it the fourth most suspicious line in the code file. Our technique also assigned a score of 0.27 on line 1, and ranks it the seventh most suspicious line in the code file. In contrast, our technique only assigned a score of 0.09 for line 13, which is not within the top 20. Language models are good at detecting these types of defects because they recognize unlikely inputs [22]. Consider just the text of the code, line 5 appears to assign a `null`-like value (the result of `getNullText()`) to `src` under some conditions. Line 7 then invokes a method on `src`. Even without knowing the implementation of `getNullText()` in depth (for which traditional program analyzers would be more suitable than language models), this pattern is suspicious to a human reader and a large language model alike.

2.2 Vulnerability Detection

Logical errors are not the only type of code mistakes that can impact software quality. Software security vulnerabilities are often the target of various forms of cyber-attacks.

The *Devign* dataset [23] labels vulnerable functions from four open-source C-language repositories (requiring 600 man hours of manual labeling). Figure 2 shows a bug from the Qemu open-source project,[3] one of the four studied repositories in *Devign*. The bug lines (highlighted) correspond to CWE-362, within the top 25 most dangerous Common Weakness Enumeration (CWE) list.[4]

```
1  DISAS_INSN(divw)
2  {
3      TCGv reg;
4      TCGv tmp;
5      TCGv src;
6      int sign;
7      sign = (insn & 0x100) != 0;
8      reg = DREG(insn, 9);
9      if (sign) {
10         tcg_gen_ext16s_i32(QREG_DIV1, reg);
11     } else {
12         tcg_gen_ext16u_i32(QREG_DIV1, reg);
13     }
14     SRC_EA(env, src, OS_WORD, sign, NULL);
15     tcg_gen_mov_i32(QREG_DIV2, src);
16     if (sign) {
17         gen_helper_divs(cpu_env, tcg_const_i32(1));
18     } else {
19         gen_helper_divu(cpu_env, tcg_const_i32(1));
20     }
21     tmp = tcg_temp_new();
22     src = tcg_temp_new();
23     tcg_gen_ext16u_i32(tmp, QREG_DIV1);
24     tcg_gen_shli_i32(src, QREG_DIV2, 16);
25     tcg_gen_or_i32(reg, tmp, src);
26     set_cc_op(s, CC_OP_FLAGS);
27 }
```

Figure 2: Qemu's CWE-362 (Race condition vulnerability)

CWE describes CWE-362 as concurrent execution using shared resource with improper synchronization (i.e., race condition). Although Qemu's repository[5] includes test cases for crashes and input behaviors, none of the test cases covers concurrency bugs that only occur during run time. Indeed, concurrency bugs like race conditions are ill-suited for discovery via traditional testing.

As a result, test based fault localization and debugging methods are clearly inapplicable to this kind of defect. This has of course motivated significant work in profiling and analysis to discover and address them [23, 24]. Chakraborty et al. [24] found that existing modeling techniques do not completely capture code semantics in vulnerability detection. Existing deep learning based vulnerability detection tools only go as far as predicting any vulnerability in a code snippet or program file, rather than individual statements. Traditional approaches such as static analysis can be used to detect race conditions [25?]. However, these approaches are either precise but not scalable or can scale for large programs but incur a high false positive rate, limiting their usage in practice.

Fortunately, a dataset like *Devign* encompasses significant manual effort in labeling existing security vulnerabilities in existing code, as has been done for lines 10, 12, and 23–25 in Figure 2.

We show in this paper that an FL-specific model pretrained on a large-scale LLM can also detect security vulnerabilities without test cases.

Our technique detects lines 3, 4, 10, 12, and 23 as faulty in Figure 2, in which lines 10, 12, and 23-25 are actual vulnerability lines. Our technique successfully localizes three of the five lines that are faulty.

[2]Note that the test suite includes another test, `testGetSetNullText`, which explicitly checks that `getNullText` returns `null` (not the empty string).
[3]https://qemu.org/
[4]https://cwe.mitre.org//top25/archive/2022/2022_cwe_top25.html
[5]https://github.com/qemu/qemu

Surprisingly, lines 3 and 4 are variable declarations (i.e., variables `reg` and `tmp`) for the actual faulty lines 10–12, and 23–25.

3 APPROACH

In this section, we discuss the key ideas behind our language model enabled fault localization technique. Figure 3 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, 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 we 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 5 shows a more detailed description of our language modeling procedure, which we describe in detail in Section 3.2. In the following sections, we describe each component of *LLMAO*.

3.1 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 [26]. 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 5 (top) shows the causal attention mechanism used to train a left-to-right language model. Figure 5 describes a simplified Transformer model for both *CodeGen* and our bidirectional language model. Auto-regressive and left-to-right LMs [13, 14, 19, 20] 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 4. 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 [13] 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. [27] for language modeling. 7.6% of the dataset is programming language data collected from GitHub repositories with >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 [14] 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 5, 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 we 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 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 left-ward context. One solution might predict buggy lines based on the final hidden state, reflecting the model's knowledge after the entire file has been processed, but this creates a bottleneck where that state must store information from each line in the entire file. This bottlenecking phenomenon [28] 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 [26].

We postulate that we 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

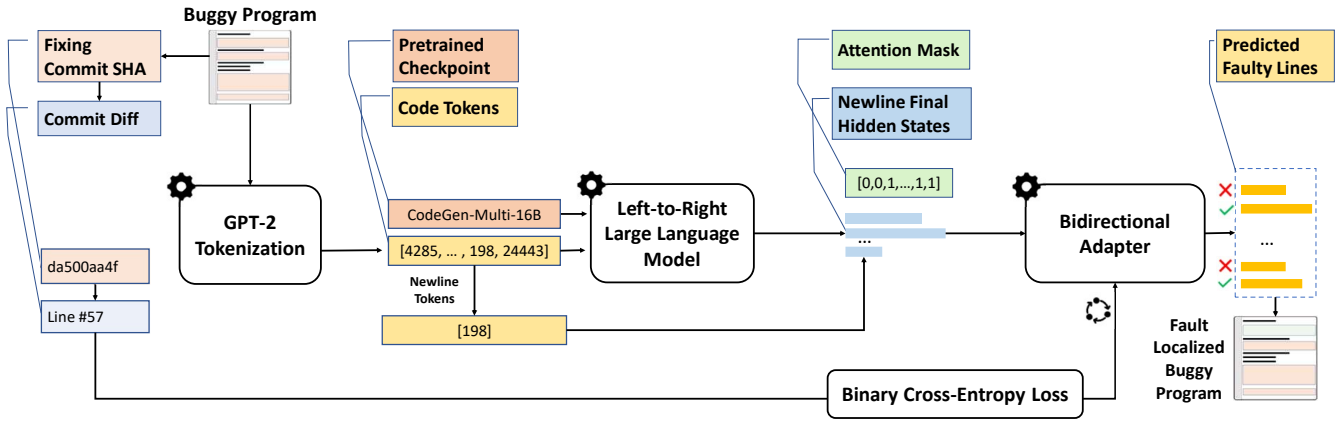


Figure 3: *LLMAO*'s architecture, which takes as input a buggy program and produces a list of suspiciousness scores for each code line

```
# Recursive binary search
def binarySearch(list, left, right, i):
    middle = ???
```

(a) Left-to-right language model prompt

```
# Recursive binary search
def binarySearch(list, left, right, i):
    middle = (left + right) // 2
    if arr[middle] == x:
        return middle
```

(b) Left to right language model completion

Figure 4: Left-to-right language model code generation

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 5, the attention masking matrix for the bidirectional model allows all tokens in the sequence to attend to each other (visualized in blue). We thus train a bidirectional adapter consisting of a configurable number of Transformer layers, following the standard Transformer encoder architecture [26]. Concretely, our approach involves five steps, visualized in Figure 5:

Step 1: We 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 we do not tune the pretrained model.

Step 2: We extract the representations of each newline token to obtain state per line in the original program: $S_{NL} \in \mathbb{R}^{M \times D} = S[c_i = \backslash n]$, 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.

Step 3: The dimension of the pretrained model's states, D, ranges up to 6,144 for the CodeGen models we 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: We 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: We transform each newline token's representation to a single value ranging from 0 to 1 via a sigmoid-activated dense projection $B = \sigma(R_{NL}\, W_b)$ where $W_b \in \mathbb{R}^{d \times 1}$. The resulting predictions per newline token can be seen as probability estimates of each line being buggy according to the model. These are compared against the ground-truth labels $T \in \{0, 1\}^M$ using the binary cross-entropy loss $\mathcal{L}_{CE} = T \ln B + (1 - T) \ln (1 - B)$. This loss is backpropagated through all layers up to, but not including, those in the pretrained network to obtain gradients. Given these gradients averaged across a sufficiently large minibatch of samples, the model states are updated to make its predictions more likely to agree with the training labels, using the setup described in Section 4.1.6.

4 EVALUATION

In this section, we present our approach and results for the following three research questions.

RQ1. How does *LLMAO* compare with prior FL techniques? We evaluate our technique's performance in comparison with existing FL techniques on the same dataset.

RQ2. How well does *LLMAO*'s performance generalize to new projects? We evaluate *LLMAO*'s performance on previously-unseen code, to assess its generalizability beyond its training data.

RQ3. How does each component of *LLMAO* impact its performance? We conduct an ablation analysis to evaluate the impact of different components on the performance of our model.

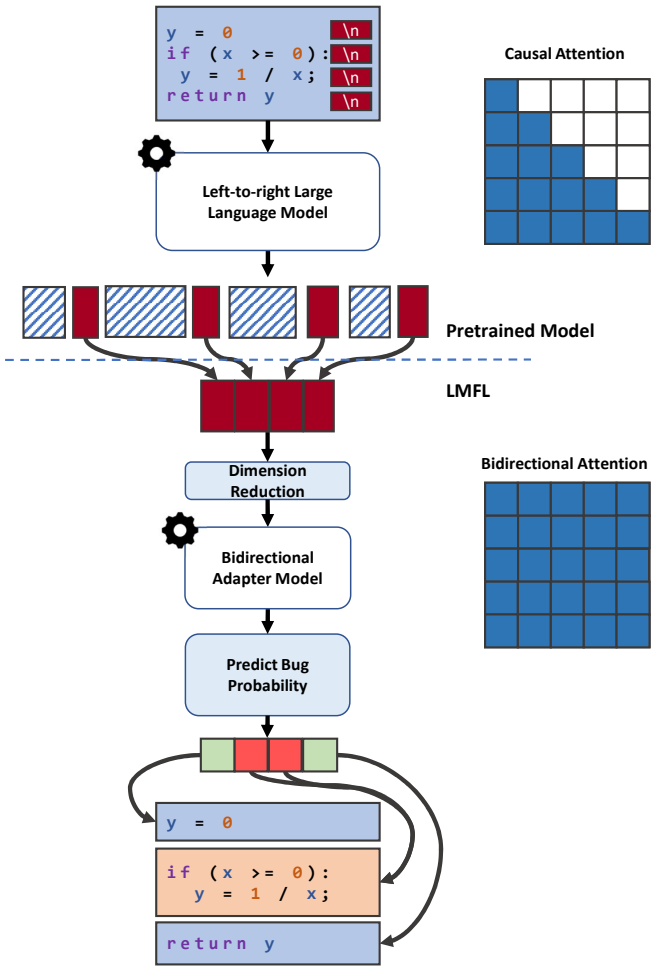


Figure 5: Attention masking procedure of *LLMAO*

RQ4. How generalizable is *LLMAO* to other languages and domains? We evaluate *LLMAO* on different languages and domains.

4.1 Setup

4.1.1 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 four datasets with source code and corresponding labeled fault lines.

- ***Defects4J* V1.2.0** : A Java benchmark dataset with 395 bugs from 6 Java projects [21]. We 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 [21]. 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 [3]).
- ***BugsInPy***: a Python benchmark with 493 bugs from 17 different projects [29].
- ***Devign*:** a C benchmark with 5,260 from two open-source projects [23]. The original Devign dataset contains 15,512 security vulnerabilities from four different projects [23]. However, the authors of Devign only released a partial dataset available online.

All datasets include fixing commits that correspond to each fault. We identify faulty statements as those that are changed in the git diff associated with each commit, following prior approaches [11, 22, 30]. We then track line numbers of changed statements as training labels.

4.1.2 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 an 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 [2], DeepRL4FL [10], and TRANSFER-FL [11]. 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. We also include Ochiai [6], the best-performing SBFL approach. We use the prior techniques' replication packages to compute Top-N scores, including their handling of tied ranks (if any); we 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 [2] 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 [10]. 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.

4.1.3 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) [2, 3, 10, 11]. An effective and robust FL tool using machine learning or

Table 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

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 overfitting to a particular project and reduces applicability, requiring a relatively rich project and bug history before a technique can be used. We 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 [2, 10], 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).

We 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.

4.1.4 Evaluation Metrics. We 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 [31], with particular attention paid to the top-5 elements [32]. To compare against state-of-the-art techniques, we adopt Top-N following prior work [2–4].

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. We 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.

4.1.5 Ablations. We conduct an ablation analysis to evaluate the impact of different components on the performance of our model (RQ3). We run five variants of our proposed technique for the *Defects4J* V1.2.0 dataset. We 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. We 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.

4.1.6 Hyperparameters. Table 1 shows the hyperparameters used in training all our models. We reduced the learning rates until both the training and validation loss converged in a stable manner. Following the established practice in language model training [33], 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 non-pretrained bidirectional model requires a much longer training time (some 2,000 epochs) for the validation loss to converge.

We 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.

4.1.7 Environment. All results presented in this section were obtained using an Intel(R) Xeon(R) 6248R CPU @ 3.00GHz running Debian GNU/Linux 1 and a single Nvidia Quadro RTX 8000 GPU. Our largest model, *LLMAO* with *CodeGen*-16B, takes 20 minutes to train on the *BugsInPy* dataset, 30 minutes on the *Defects4J* dataset, and 2 hours on the *Devign* dataset.

4.2 Results

RQ1: How does *LLMAO* compare with prior DL-based FL tools? Table 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; we evaluate our *LLMAO* using three *CodeGen* pretrain sizes. The results show the Top-N ($N \in \{1, 3, 5\}$) score for each technique. Table 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 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).

Table 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%)
LMFL 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%)

A Wilcoxon signed-rank test [34] 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 [2]).

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? We 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 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 we 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 [3] 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.

Table 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%)

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 2 show the impact of pretrained models on *LLMAO*'s performance.

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 We 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, an 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. We perform an additional Wilcoxon signed-rank test [34] 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.

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* [29] dataset for localizing Python bugs, and the *Devign* [23] 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.

We observe from Table 3 that *LLMAO* can localize faulty statements with Top-1 of 10.3% on *BugsInPy*, and 28.1% for *Devign*. We 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 6a, 6b and 6c 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. We 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.

We observe a clear improvement on the AUC as we use the *CodeGen* final hidden states for training, and the AUC continues to improve as we use larger *CodeGen* models. In particular, the AUC for Figure 6a 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 6b and 6c show a significant improvement in our model's predictive power as we 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

Our approach generalizes to other languages and domains, given a large enough labeled. *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.

5 RELATED WORK

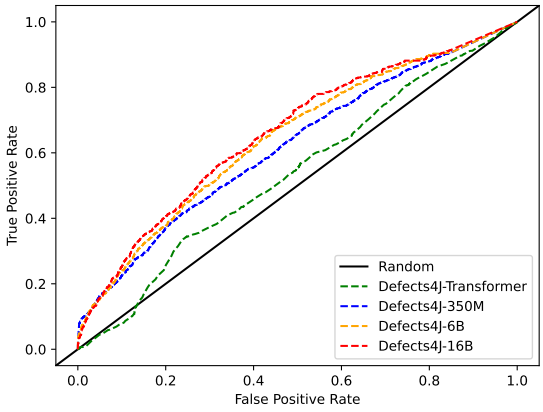
We discuss in the following sections the most recent advances in fault localization and LLM for code.

5.1 SBFL and MBFL

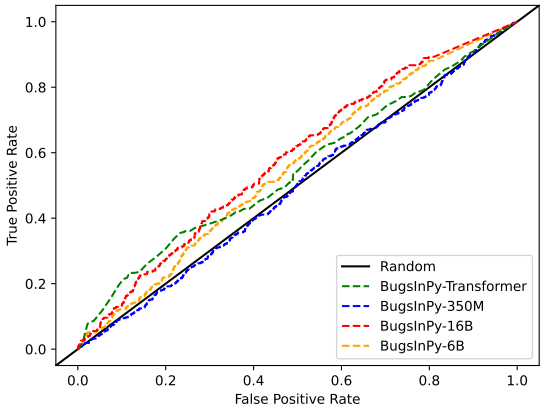
Spectrum-based Fault Localization (SBFL) [1, 6, 35, 36] and Mutation-based Fault Localization (MBFL) [8, 37–39] have been extensively studied for fault localization.

SBFL calculates the suspiciousness score of each code line to represent the probability of the line being faulty. SBFL measures the number of failed and passed tests that cover each code line to generate the suspiciousness score. To generate suspiciousness scores, SBFL uses a ranking formulae based on the test coverage information of each code line. Although SBFL is widely accepted due to its simplicity and efficiency, SBFL takes coverage as the only input information, and more specifically only retrieves the number of tests from test coverage information. Test coverage alone cannot always encapsulate the faulty behaviors from code lines.

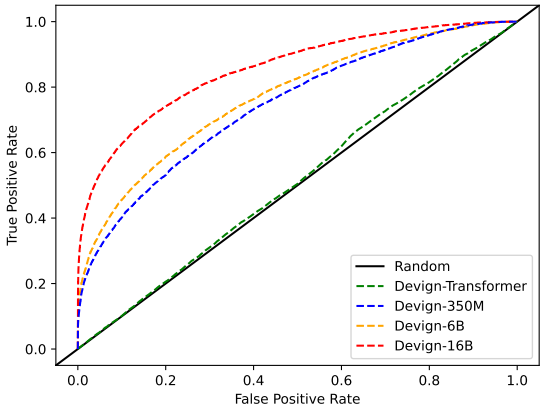
MBFL techniques [8, 37–39] mitigate the limitations of SBFL by applying mutation testing to generate mutants for the original program under a test suite. The original program is mutated with



(a) ROC curves on the *Defects4J* dataset



(b) ROC curves on the *BugsInPy* dataset



(c) ROC curves on the *Devign* dataset

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

one syntactic change that corresponds to a predefined rule. The mutation rules are called mutation operators, (e.g., change if (a==b) into if (a!=b)). MBFL techniques use mutants to check the impacts of code elements on the test outcomes for fault localization. MBFL techniques consider the impact information whereas SBFL does not. However, MBFL techniques would fail in cases where an element does not have a possible mutant for impact simulation.

Although leveraging test coverage and program mutation is relatively simpler than billion parameter deep learning models, SBFL cannot accurately rank the statements with the same spectrum scores, and MBFL fails in situations where a mutant cannot be instantiated. Furthermore, both SBFL and MBFL depend on the extent the test suite can soundly and completely cover the source code. Our work does not depend on test coverage, but instead uses the naturalness of code [22].

5.2 MLFL

Machine learning fault localization techniques have been proposed using insights from program analysis on code behavior. Prior MLFL techniques train on data such as test coverage [40, 41], co-changing method declaration and corresponding statement-level calls [4], or the program's code structure, such as the abstract syntax tree (AST) [10]. Recent deep learning techniques, such as GRACE [3] and FixLocator [4], encode the AST and test coverage as graph representations and learn to rank faulty methods with graph neural networks (GNN) [42]. GNNs can directly analyze graph structured information with all topological dependencies reserved, so as to not lose information through data preprocessing. DeepRL4FL [10] use Convolution Neural Network (CNN) [43] applied on code coverage (CC) matrix. TRANSFER-FL [11] leverages the deep semantic features and transferred knowledge from open-source data to improve fault localization. DeepFL [2] and TRANSFER-FL [11] combine semantic-based, spectrum-based, and mutation-based features and use a multi-layer perceptron (MLP) model for fault localization.

In contrast, *LLMAO* does not require test code, an AST parser, and includes a text tokenizer and embedding layer by construction. Prior MLFL models base their architecture on recurrent or convolutional neural networks. *LLMAO* leverages attention mechanisms [26] and build bidirectional adapter layers on pretrained left-to-right LLMs directly on source code.

5.3 LMs for Code

Language models (LMs) are widely applied to natural language [44]. Recent advances in language modeling using code as training data have shown that LMs can perform code completion [45] and generate code based on natural language [46] with impressive results. Large Language Models (LLMs) have drastically raised performance on these tasks [14]. However, studies have shown that LLM code generation techniques, such as Codex [14] or GPT-Neo [19], can be prompted to generate buggy programs, including ones with security vulnerabilities [47]. Furthermore, many existing code based LLMs are not publicly available for customization for specific tasks [17]. Our work shows how to build a bidirectional language model fine-tuned on a left-to-right model, and trained specifically for the task of fault localization.

6 DISCUSSION AND THREATS

Why does it work? Recent LLMs train on such a large corpus of data that they can generate functionally correct code bodies from simple natural language documentation [14].

Models such as Codex [14] and InCoder [48] can perform the opposite direction as well: generate natural language docstring from code snippets alone. These abilities suggest that LLMs extract a significant amount of semantic knowledge from the code they process, even as their objective is just to predict each next token.

We believe that this ability to reason about code semantics translates naturally to reasoning about defects or vulnerabilities. While the model may have been mostly trained on correct code, it likely notes surprising, bug-related patterns much as traditional language models do [22], incorporating this information into token representation, since knowledge of a potential mistake is important for next-token prediction as well. Our first attempt to directly train a fault localizer on top of an LLM specialized in program synthesis supported this notion by yielding surprisingly strong results (Top-5 score of 85), but still fell short of common baselines. Our second key observation is that the LLM's knowledge at any given token is also incomplete, lacking awareness of suffix. Many bugs, including the one in Figure 1a, cannot be reliably determined until close to the end of the program, so the model is incentivized to store information about potentially important missing knowledge (in the example: whether `getNullText()` can return `null`) in the representation of earlier tokens (those on line 5) for later tokens to consider. Our adapter layers were subsequently introduced to exploit the observation that the representations of later tokens might contain valuable information for determining the bugginess of earlier ones. The resulting model outperformed all baselines and showed strong signs of improving with the scale of the underlying LLM, strongly supporting this approach.

Threats to validity. One threat to *internal validity* lies in using `diff` descriptions of bug-fixing commits to identify faulty statements. Different annotators can disagree about the true cause of a defect [49], and parsing commits provides a noisy proxy for truly faulty lines associated with a bug. We mitigate this threat first by relying on well-established previously-published datasets with historical bug fixes. These datasets are manually curated to confirm that the commits in question do fix a given bug, and *Defects4J*'s bug-fixing commits are further pruned to include only bug-relevant changes (reducing the influence of unrelated "tangled" changes such as refactorings). We note that *Defects4J*, on which the bulk of our experiments are performed, is a very common dataset in prior fault localization work, supporting comparison and consistency. The use of the commit to indicate ground-truth faulty statements or methods has similar precedent in the literature (e.g., but not limited to, refs [11, 22, 30]). At worst, a developer fix provides a conservative approximation of code defectiveness. We further mitigate the risks of mistakes in our implementation by releasing our scripts, code, and data as part of a replication package for this work available at https://figshare.com/s/35f36bff735d3c805a89. Threats to *external validity* lie in whether results on our benchmarks will generalize to real-world contexts. To reduce this threat, we evaluate on the widely-used *Defects4J*-V1.2.0 [21] with hundreds of real-world bugs. Although it is a widely-used benchmark, recent techniques may be overfitting to it [50]; we therefore additionally train our tool on two other benchmarks, *BugsInPy* and *Devign*.

As *LLMAO* is trained on top of *CodeGen*, which takes as training data roughly 65 GiB of code from GitHub repositories up to 2021, we can not fully mitigate the bias that our datasets could be included in its training data. However, *CodeGen* was trained on a very large volume of code, meaning that the model retains relatively little memory of our partition of that training data. The training data of *CodeGen* also does not include the manually annotated labels for bugs, but rather the repositories of the faulty code directly, which would not bias our fault localization models.

Threats to *construct validity* lie in measurements used. We use multiple metrics widely used to evaluate both fault localization and ML models. We also perform our experiments under 10-fold cross validation and across three datasets to strengthen generalizability.

7 CONCLUSIONS

In this paper, we propose *LLMAO*, an LLM-based approach for localizing program defects, which include general logic defects as well as security vulnerabilities. We perform an empirical study on 395 real bugs from *Defects4J*, 493 bugs from *BugsInPy*, and 5,260 security vulnerabilities from *Devign*. Our results show that *LLMAO* can outperform existing state-of-the-art deep learning based fault localization techniques without the use of insights from extensive program analysis, or any test cases. In particular, *LLMAO* can localize 48/395 more faults within the Top-5 than the first proposed deep learning based fault localizer, DeepFL, which is guided by SBFL and MBFL artifacts that require extensive manual labor to attain. *LLMAO* can localize 23/155 more bugs within Top-5 than TRANSFER-FL, which is the latest state-of-the-art deep learning based fault localizer. The comparison of AUC on different versions of our model shows that training on top of larger LLMs improves performance significantly and that our approach based on bidirectional adapter layers is essential for achieving state-of-the-art localization scores. The experimental results show that pretraining on the largest *CodeGen* model (e.g., 16 billion parameters) achieves the highest AUC on all our studied datasets. To the best of our knowledge, *LLMAO* is the first DL based tool to localize security vulnerabilities on a line level without requiring test cases or even compilable code.

ACKNOWLEDGEMENTS

This work was partially supported by the US National Science Foundation (NSF) awards CCF-1750116 and CCF-1762363, and by ANI 045917 award funded by FEDER and Portuguese Foundation for Science and Technology (FCT).

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98, IEEE, 2007.

[2] X. Li, W. Li, Y. Zhang, and L. 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*, pp. 169–180, 2019.

[3] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. 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, pp. 664–676, 2021.
[4] Y. Li, S. Wang, and T. 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*, pp. 659–671, 2022.
[5] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273–282, 2005.
[6] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46, IEEE, 2006.
[7] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 765–784, 2013.
[8] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 153–162, IEEE, 2014.
[9] T. T. Chekam, M. Papadakis, and Y. Le Traon, "Assessing and comparing mutation-based fault localization techniques,"
[10] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 661–673, IEEE, 2021.
[11] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1169–1180, 2022.
[12] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7, 2022.
[13] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.
[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
[15] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.
[16] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.
[17] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.
[18] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 511–523, 2022.
[19] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, *et al.*, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022.
[20] L. Tunstall, L. Von Werra, and T. Wolf, *Natural language processing with transformers*. O'Reilly Media, Inc., 2022.
[21] R. Just, D. Jalali, and M. 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*, pp. 437–440, 2014.
[22] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 428–439, 2016.
[23] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.
[24] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.
[25] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 237–252, 2003.
[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
[27] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.
[28] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.
[29] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. 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*, pp. 1556–1560, 2020.
[30] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, oct 2019.
[31] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proceedings of the 2011 international symposium on software testing and analysis*, pp. 199–209, 2011.
[32] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 165–176, 2016.
[33] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, *et al.*, "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.
[34] R. F. Woolson, "Wilcoxon signed-rank test," *Wiley encyclopedia of clinical trials*, pp. 1–3, 2007.
[35] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference*, vol. 1, pp. 449–456, IEEE, 2007.
[36] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance*, pp. 23–32, IEEE, 2011.
[37] V. Musco, M. Monperrus, and P. Preux, "A large-scale study of call graph-based impact prediction using mutation testing," *Software Quality Journal*, vol. 25, pp. 921–950, 2017.
[38] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.
[39] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *2010 IEEE international conference on software maintenance*, pp. 1–10, IEEE, 2010.
[40] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *The 18th IEEE International Symposium on Software Reliability*, pp. 137–146, IEEE, 2007.
[41] Z. Zhang, Y. Lei, Q. Tan, X. Mao, P. Zeng, and X. Chang, "Deep learning-based fault localization with contextual information," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 12, pp. 3027–3031, 2017.
[42] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.
[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.
[44] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," *Advances in neural information processing systems*, vol. 13, 2000.
[45] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 345–356, 2016.
[46] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pp. 419–428, 2014.
[47] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "An empirical cybersecurity evaluation of github copilot's code contributions," *ArXiv abs/2108.09293*, 2021.
[48] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.
[49] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 117–128, Association for Computing Machinery, 2017.
[50] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302–313, 2019.

Modelos tradicionais não detectam vulnerabilidades de segurança.

Fault Localization (FL), Left to right Language Model LLM

Existem técnicas de "Fault Localization" como o: SBFL e MBFL e, usando o Machine Learning, MLFL, porém as técnicas que usam Machine Learning ainda possuem limitações: depender dos escores das outras técnicas, depender da cobertura dos testes, escrita da esquerda para a direita.