# Large Language Models for Test-Free Fault Localization

서울시립대학교
UNIVERSITY OF SEOUL

CIDA Lab.

전자전기컴퓨터공학부
2022440119 전형준

2025.01.02

Table of Contents

# Fault Localization

Fault localization (FL) approaches aim to automatically identify which program entities (like a line, statement, module, or file) are implicated in a particular bug

```
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 }
```

# SBFL/MLFL

Spectrum based fault localization (SBFL) approaches, such as Tarantula or Ochiai, apply statistical analysis on the coverage data of failed/passed tests to compute the suspiciousness of code elements.

$$Ochiai(\ell) = \frac{\ell_f}{\sqrt{(\ell_f + \ell_p) \cdot (\ell_f + n_f)}}$$

$\ell$ f : The number of failing tests that cover the code line $\ell$.

$\ell$ p : The number of passing tests that cover the code line $\ell$.

n f : The number of failing tests that do not cover the code line $\ell$.

# SBFL/MLFL

```
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  }
```

# SBFL/MLFL

Recent advances in Machine learning based fault localization (MLFL), like DeepFL, DEEPRL4FL, and GRACE use machine learning to relate code, test, or execution features to the likelihood of faultiness for a program entity

DeepRL4FL use Convolution Neural Network (CNN) applied on code coverage (CC) matrix

DeepFL and TRANSFER-FL combine semantic-based, spectrum-based, and mutation-based features and use a multi-layer perceptron (MLP) model for fault localization

# LLMAO

Recent LLMs train on such a large corpus of data that they can generate functionally correct code bodies from simple natural language documentation

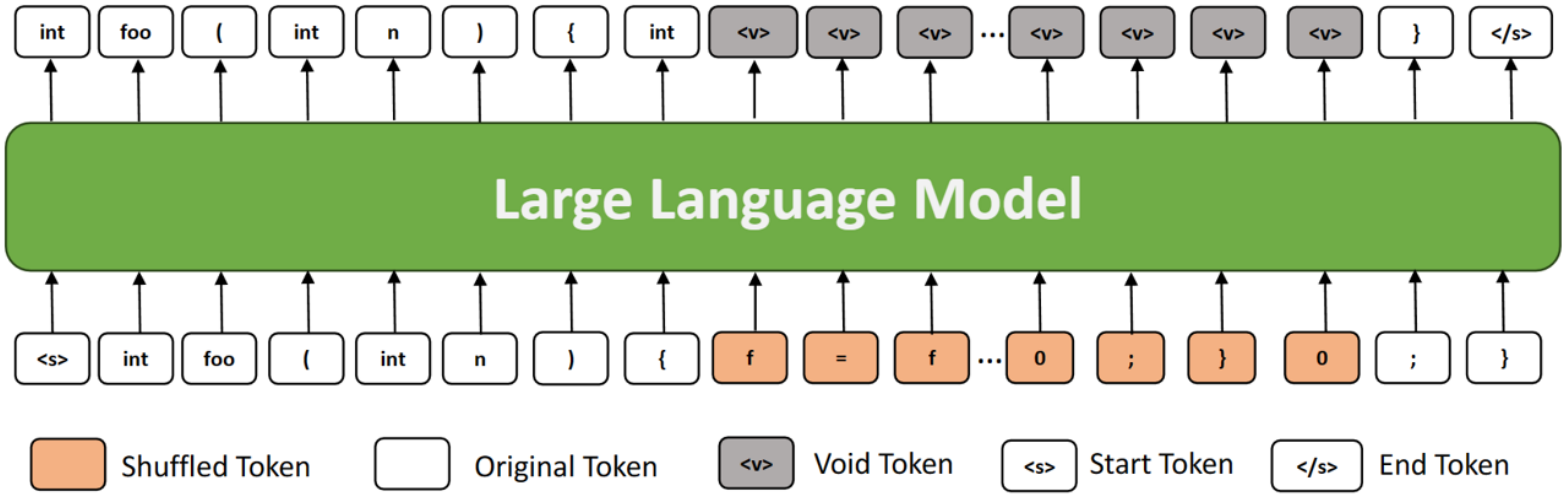
Models such as Codex and InCoder can perform the opposite direction as well: generate natural language docstring from code snippets alone.

↓

These abilities suggest that LLMs extract a significant amount of semantic knowledge from the code they process

# LLMAO

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




https://arxiv.org/html/2404.08885v1

# LLMAO

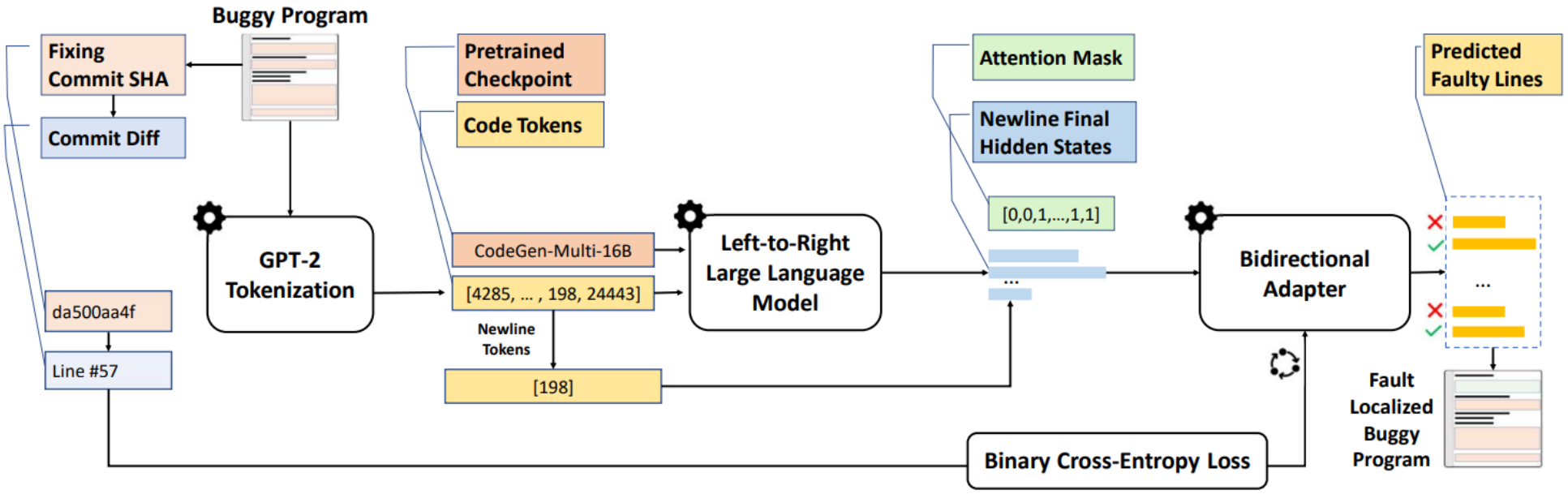



**Figure 3:** *LLMAO*'s architecture, which takes as input a buggy program and produces a list of suspiciousness scores for each code line

# LLMAO

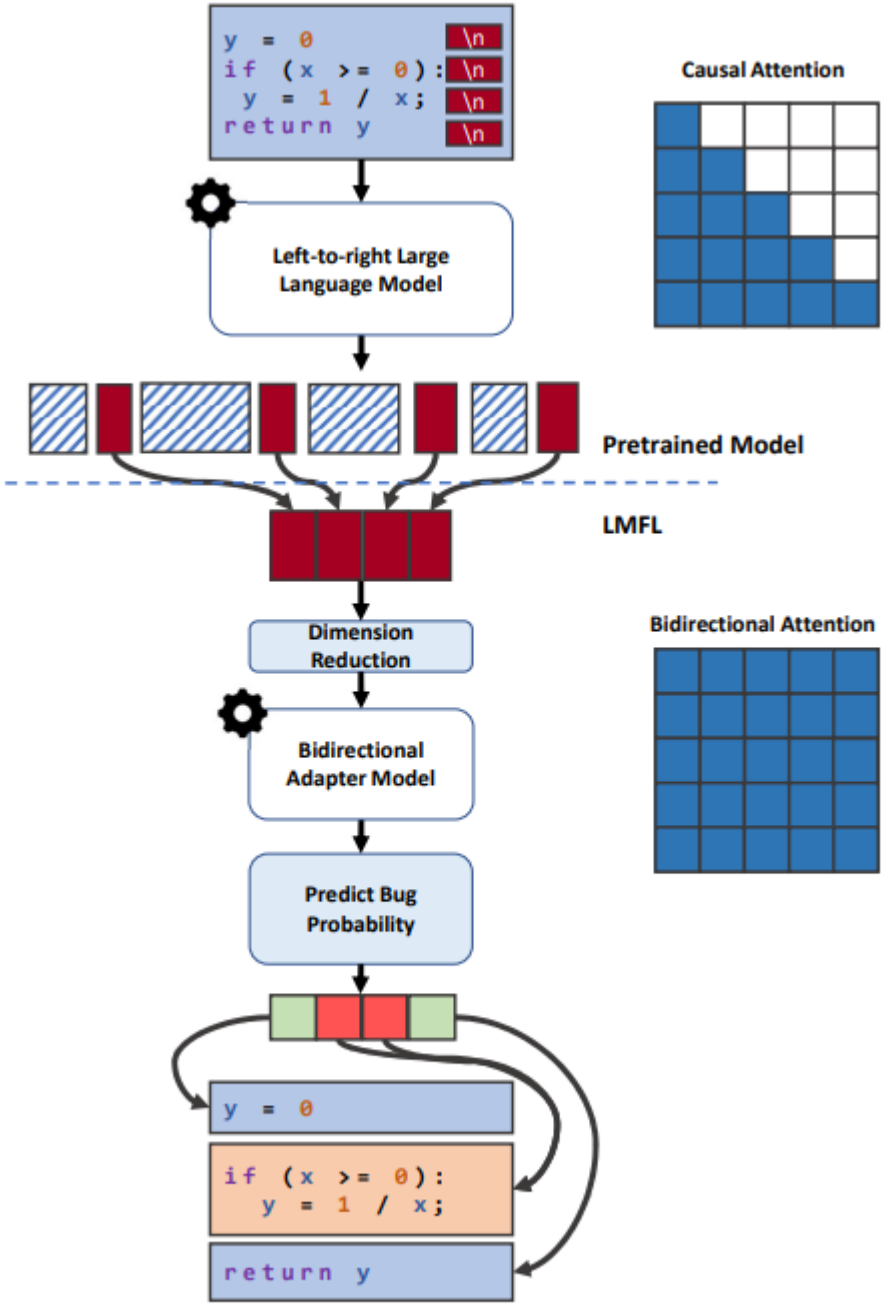
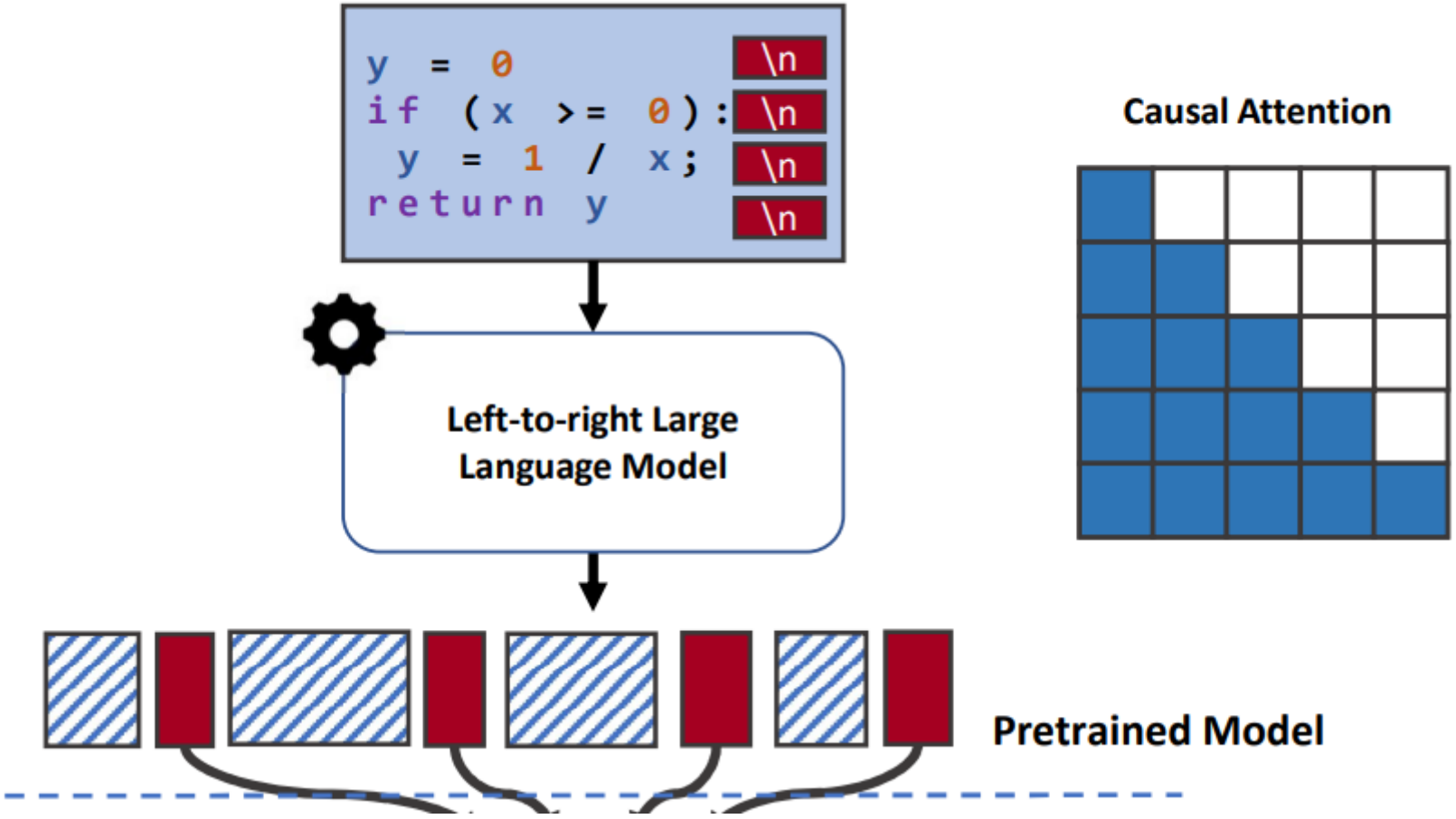
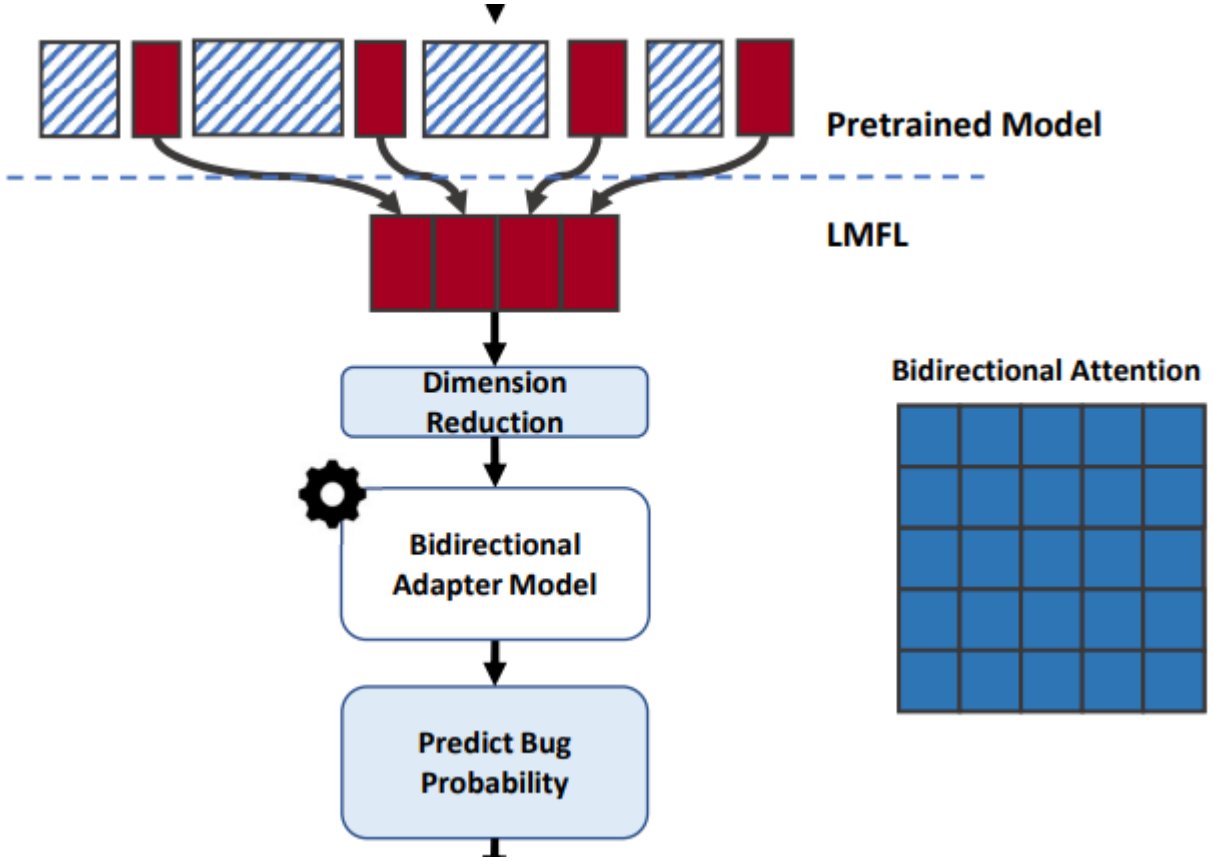



**Figure 5: Attention masking procedure of *LLMAO***

# LLMAO

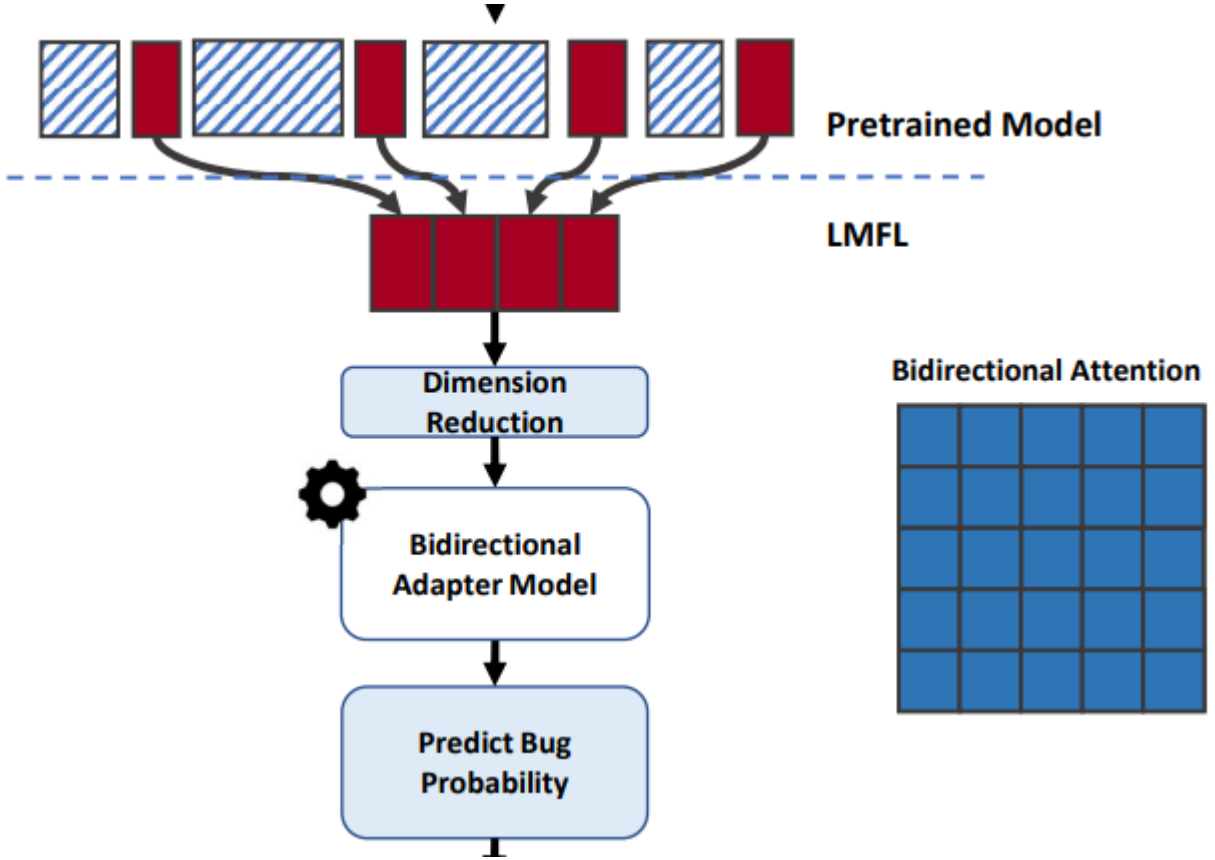

y = 0 \n
if (x >= 0): \n
y = 1 / x; \n
return y \n
Left-to-right Large Language Model
Causal Attention
Pretrained Model

# LLMAO

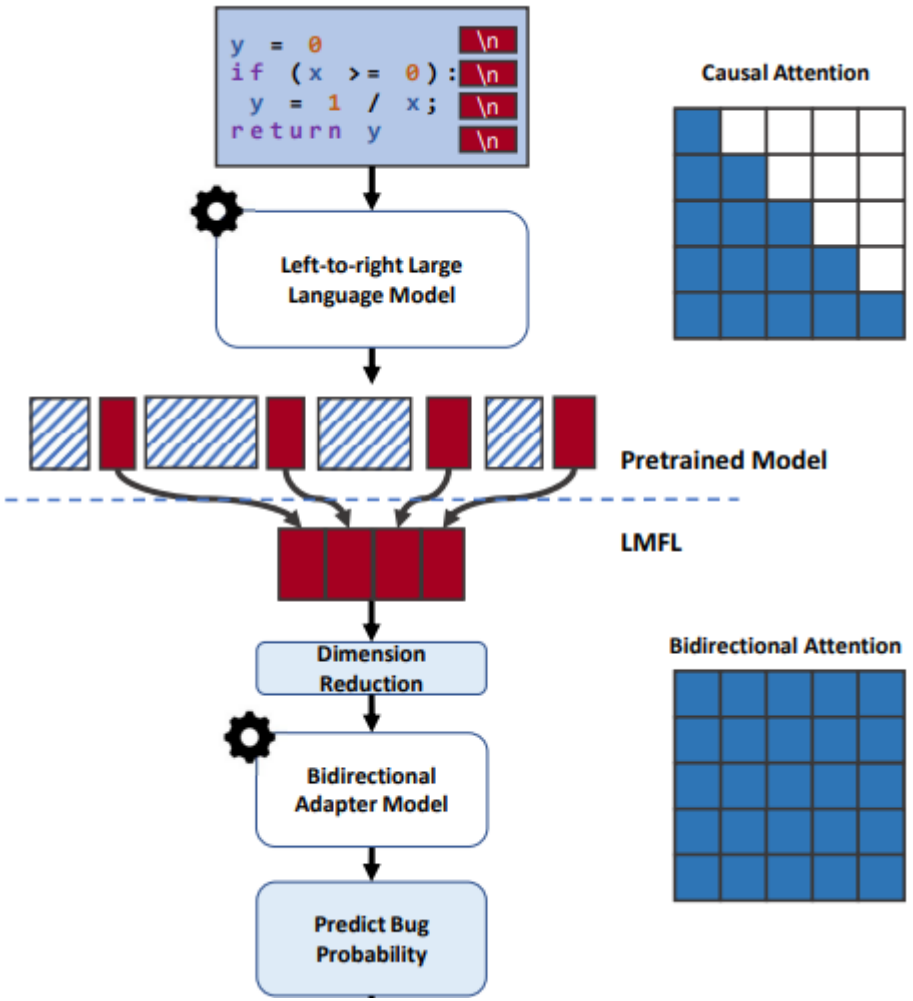



We extract the final hidden states for each newline (NL) token in each training sample from CodeGen to produce a condensed sequence representation in which each token represents one line.

# LLMAO




By removing the causal attention mask that normally prevents the exchange of information with “future” tokens in our added layers…










$$C = [c_0, c_1, \ldots, c_N]$$

$$S \in \mathbb{R}^{N \times D}$$

$$S_{NL} \in \mathbb{R}^{M \times D}$$

$$R_{NL} \in \mathbb{R}^{M \times d} = S_{NL}\, W_d \; W_d \in \mathbb{R}^{D \times d}$$

$$A_{NL} \in \mathbb{R}^{M \times d}$$

$$B = \sigma(R_{NL}\, W_b) \quad W_b \in \mathbb{R}^{d \times 1}.$$

# LLMAO

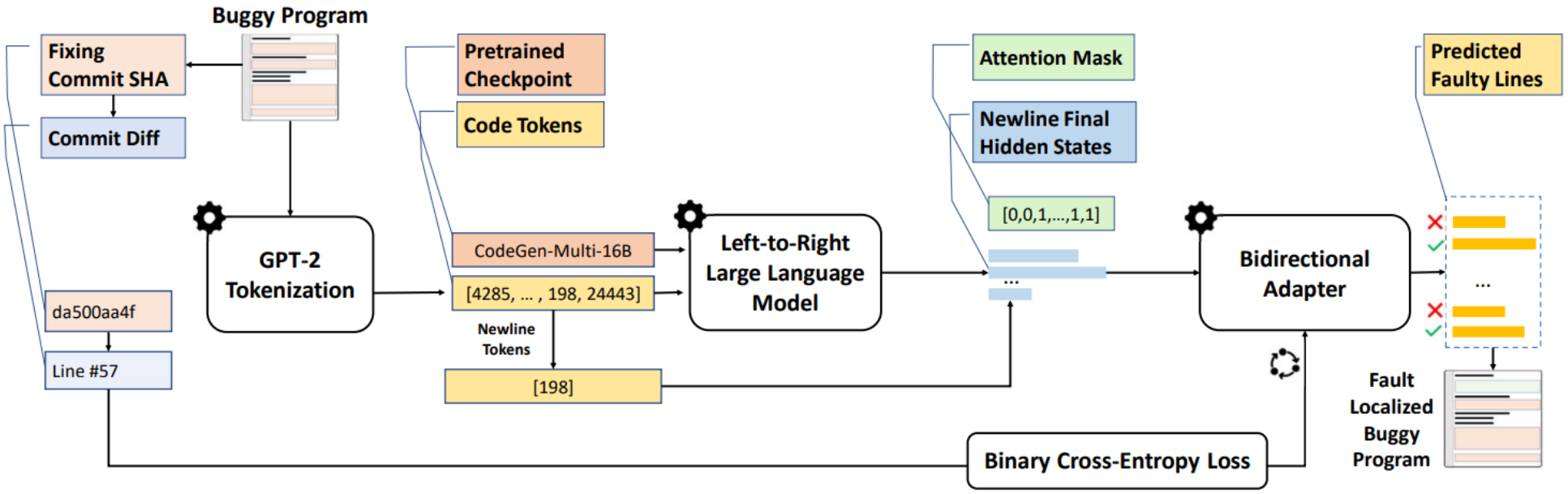



**Figure 3:** *LLMAO*'s architecture, which takes as input a buggy program and produces a list of suspiciousness scores for each code line

# Dataset

Defects4J V1.2.0 : A Java benchmark dataset with 395 bugs

We use V1.2.0 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. 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

BugsInPy: a Python benchmark with 493 bugs from 17 different projects.

Devign: a C benchmark with 5,260 from two open-source

# Evaluation & Experiments

RQ1: How does LLMAO compare with prior DL-based FL tools?

**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%)** |

Both DeepFL and TRANSFER-FL techniques include mutation-based fault localization information, which is very time-consuming to collect

# Evaluation & Experiments

RQ2. How well does LLMAO ’s performance generalize to new projects?

| | | | | |
|---|---|---|---|---|
| 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%) |

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)

# Evaluation & Experiments

RQ3. How does each component of LLMAO impact its performance?

| | | | | |
|---|---|---|---|---|
| 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%) |

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.

# Evaluation & Experiments

RQ4. How generalizable is LLMAO to other languages and domains?

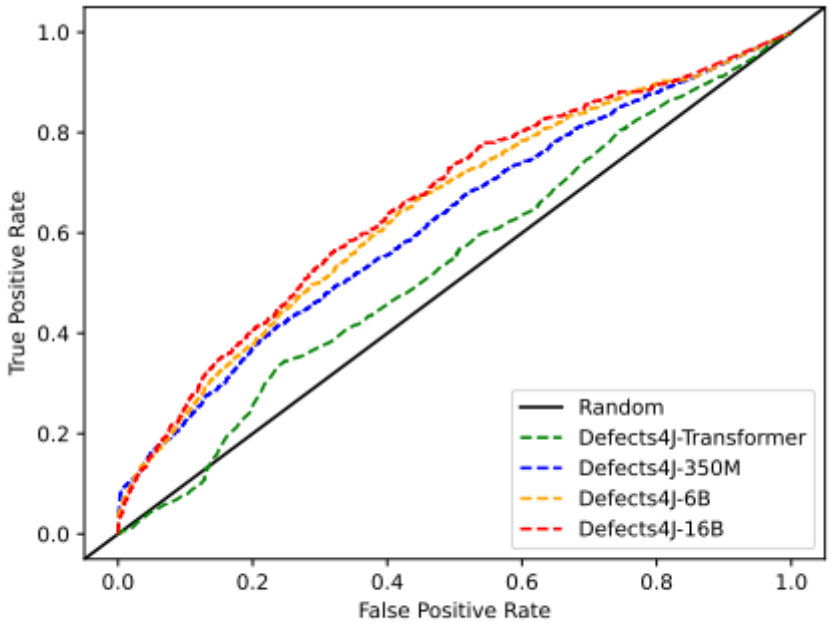
| Metric | *BugsInPy* | *Defects4J* | *Devign* |
|---|---|---|---|
| # lines | 76,672 | 168,960 | 7,180,160 |
| Top-1 | 51/493 (10.3%) | 88/395 (22.3%) | 1478/5260 (28.1%) |
| Top-3 | 59/493 (12.0%) | 149/395 (37.7%) | 2050/5260 (39.0%) |
| Top-5 | 75/493 (15.2%) | 183/395 (46.3%) | 3171/5260 (60.3%)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

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.

# Evaluation & Experiments

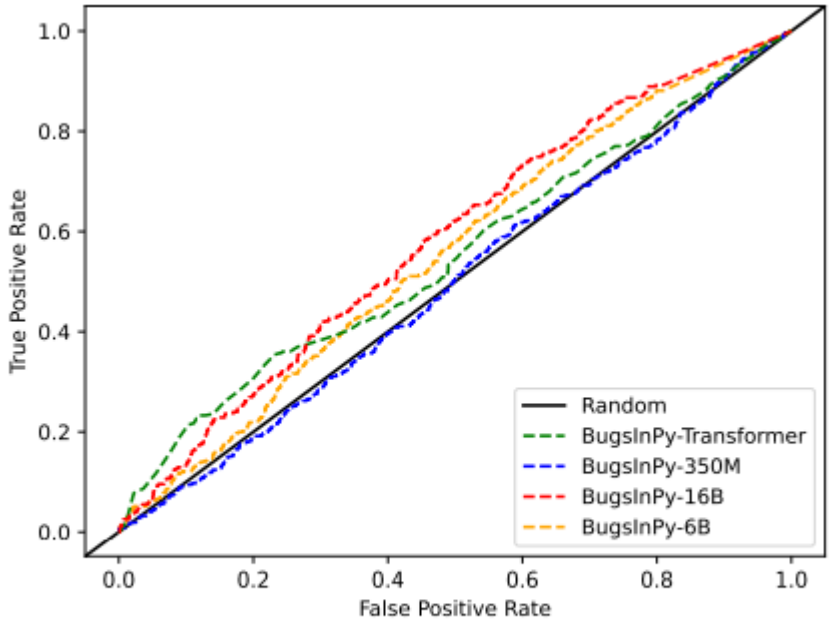
RQ4. How generalizable is LLMAO to other languages and domains?



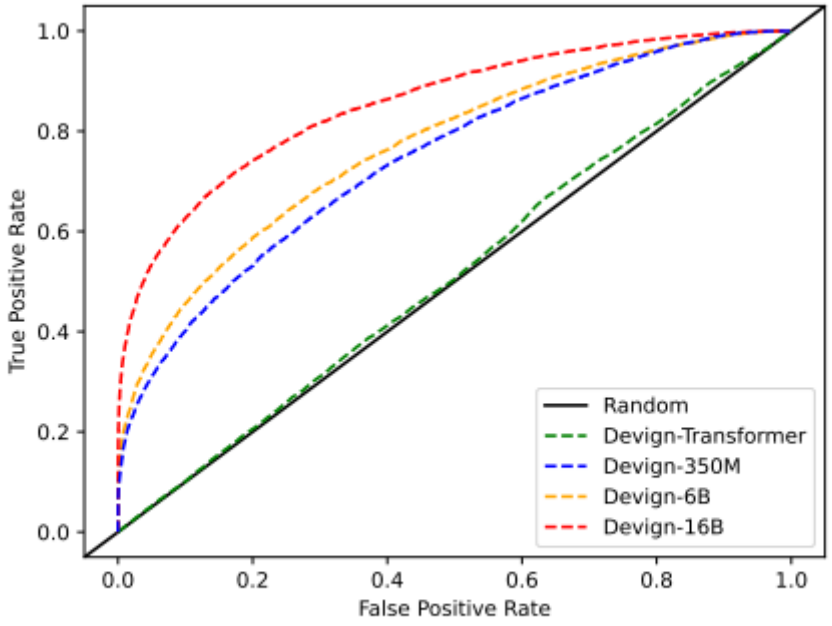

**(a) ROC curves on the *Defects4J* dataset**

# Evaluation & Experiments

RQ4. How generalizable is LLMAO to other languages and domains?




**(b) ROC curves on the *BugsInPy* dataset**




**(c) ROC curves on the *Devign* dataset**

# Conclusion

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