Learning to Detect and Localize Multilingual Bugs

HAORAN YANG, Washington State University, USA
YU NONG, Washington State University, USA
TAO ZHANG, Macau University of Science and Technology, China
XIAPU LUO, Hong Kong Polytechnic University, China
HAIPENG CAI*, Washington State University, USA

Increasing studies have shown bugs in multi-language software as a critical loophole in modern software quality assurance, especially those induced by language interactions (i.e., *multilingual bugs*). Yet existing tool support for bug detection/localization remains largely limited to single-language software, despite the long-standing prevalence of multi-language systems in various real-world software domains. Extant static/dynamic analysis and deep learning (DL) based approaches all face major challenges in addressing multilingual bugs. In this paper, we present xLoc, a DL-based technique/tool for detecting and localizing multilingual bugs. Motivated by results of our bug-characteristics study on top locations of multilingual bugs, xLoc first learns the *general knowledge* relevant to differentiating various multilingual control-flow structures. This is achieved by pre-training a Transformer model with *customized position encoding* against *novel objectives*. Then, xLoc learns *task-specific knowledge* for the task of multilingual bug detection/localization, through *another new position encoding scheme* (based on cross-language API vicinity) that allows for the model to attend particularly to control-flow constructs that bear most multilingual bugs during fine-tuning. We have implemented xLoc for Python-C software and curated a dataset of 3,770 buggy and 15,884 non-buggy Python-C samples, which enabled our extensive evaluation of xLoc against two state-of-the-art baselines: fine-tuned CodeT5 and zero-shot ChatGPT. Our results show that xLoc achieved 94.98% F1 and 87.24%@Top-1 accuracy, which are significantly (up to 162.88% and 511.75%) higher than the baselines. Ablation studies further confirmed significant contributions of each of the novel design elements in xLoc. With respective bug-location characteristics and labeled bug datasets for fine-tuning, our design may be applied to other language combinations beyond Python-C.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Multi-language software, multilingual bugs, bug detection, fault localization

ACM Reference Format:
Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to Detect and Localize Multilingual Bugs. *Proc. ACM Softw. Eng.* 1, FSE, Article 97 (July 2024), 24 pages. https://doi.org/10.1145/3660804

1 INTRODUCTION

A number of prior studies [6, 8, 33, 45, 46, 49, 59, 76] have revealed that software developed in multiple, interacting programming languages (i.e., *multi-language software*[1]) is prevalent, both

*Haipeng Cai is the corresponding author.
[1]They are also known as *multilingual software*. Note that systems merely using multiple languages in which the languages do not have interactions/interoperations are *trivially* multilingual, which are out of our scope/interest in this paper.

Authors' addresses: Haoran Yang, Washington State University, Pullman, USA, haoran.yang2@wsu.edu; Yu Nong, Washington State University, Pullman, USA, yu.nong@wsu.edu; Tao Zhang, Macau University of Science and Technology, Macau, China, tazhang@must.edu.mo; Xiapu Luo, Hong Kong Polytechnic University, Hong Kong, China, csxluo@comp.polyu.edu.hk; Haipeng Cai, Washington State University, Pullman, USA, haipeng.cai@wsu.edu.

ACM 2994-970X/2024/7-ART97
https://doi.org/10.1145/3660804

in industry [25] and the open-source world [13, 62], and their dominance (over single-language software) continues to grow [34, 45, 63]. Intuitively, this prevalence and dominance is justifiable given the impetus (e.g., benefits or even necessity) of multilingual software development [1, 8, 75]. Indeed, different languages have their own peculiar strengths and weaknesses [11, 49]. Thus, combining various languages could be a natural consideration by developers for building software that requires capabilities each best offered by one of the languages [49, 64].

Meanwhile, compared to single-language systems, multilingual software is generally much more complex [2, 20], not just due to the sheer aggregation of complexities from individual languages used, but more because of the complications from interoperations among heterogeneous *language units* (i.e., code components each written in one single language) [4, 46]. Intuitively, the greater complexity leads to potentially more bugs in multi-language software [20, 26], especially those *induced by cross-language information flow as a result of language interfacing/interactions* (i.e., *multilingual bugs*) [29–31]. Multi-language software is also subject to bugs contained in each language unit alone [31, 35]. Yet those bugs can be addressed by the respective single-language testing/debugging tools, which are richly available for many languages. By contrast, multilingual bugs are more of a *loophole* in testing and fault diagnosis of modern software systems (which are mostly multilingual).

In response, techniques that aspire to address this loophole have emerged, including those based on static/dynamic analysis and deep learning (DL) [7, 24, 29, 30, 35, 36, 74]. *Static techniques*, however, generally suffer from excessive impression [70] in addition to unsoundness due to the prevalence of dynamic language constructs in real-world programs [30]. In particular, to overcome the semantics disparity of different languages [4], existing static techniques for multi-language software rely on heavy language-specific analyses (e.g., computing abstract summaries of native code [29, 55] or defining semantic rules for each interoperation API [77]), which requires much engineering work for each particular language combination hence is difficult to extend to other combinations. *Dynamic techniques* can be much more precise, but they rely on existing tests which are usually limited in terms of code coverage [5, 30, 72]. Moreover, test cases are commonly lacking, especially for real-world multi-language systems [35]. Generating more tests helps, but it either needs quite some manual work [24] or often ends up finding no bugs after extensive exploration due to the random nature of the technique [35]. Also, some of these dynamic techniques [14, 15, 17, 22, 42] essentially only test one language unit at a time, not directly addressing multilingual bugs.

In this context, *DL-based approaches* have gained increasing momentum in recent years. Leveraging existing bug datasets, these data-driven techniques avoid much of the limitation of static/dynamic techniques. They also tend to more extensible to various language combinations, as they do not have to rely on much language-specific analysis. However, even with respect to the rich relevant literature, developing a DL-based technique that effectively detects and localizes multilingual bugs faces at least two *key challenges*. First, DL-based fault localization techniques commonly rely on (failed) test cases that can reveal bugs [78]. These test executions provide essential guidance for the DL model to (learn to) differentiate buggy from non-buggy code entities [27, 36–38, 44, 48]. However, as noted earlier, test cases are often lacking for multi-language software in the wild [35] and those that reveal multilingual bugs are rarely available. Without the guidance, learning to locate bugs based on code alone is much more difficult (**Challenge 1**). Second, building an effective DL model typically needs a large set of (quality) training (especially labeled buggy) samples [51, 53], which are unavailable. Fine-tuning a pre-trained model demands much less task-specific samples, which is also a popular approach adopted by numerous DL-based software engineering techniques [69]. However, even if we can manually curate such a task-specific dataset, it is most likely to be small due to the great cost of manual labeling. For effectively fine-tuning with such a small dataset, the pre-trained model would need to have *general knowledge that is relevant to the downstream task* (i.e., detecting/localizing multilingual bugs in our case). Yet such a pre-trained model does not exist and it is unclear how to learn the general knowledge for our task (**Challenge 2**).

To address these challenges, we developed xLoc, a DL-based technique that aims to detect and localize multilingual bugs based solely on the source code of multi-language software. Immediately informed by our study on the characteristics of 200+ real-world multilingual bugs, especially concerning where those bugs are located, xLoc is enabled by three major *novel design elements*. First (**Design 1**), to overcome the lack of test-execution guidance, xLoc customizes the Transformer model [65] to attend to different control flow structures of multilingual code. The attentions serve as guidance to help the model learn to ***generally*** *detect (a buggy function) and further localize (buggy statements in the buggy function)*, realized via control-flow-based *absolute* and *relative* position encoding, respectively, hence addressing *Challenge 1*. Second (**Design 2**), to actually learn the general knowledge, xLoc pre-trains the customized Transformer against two novel pre-training objectives that supervise the model to attend to control flow constructs that tend to bear most multilingual bugs, on a large code corpus of each of the languages used. Third (**Design 3**), to fine-tune the pre-trained model for our downstream task, xLoc follows the similar insights for the pre-trained model customization but computes the absolute and relative position encoding so as to help the model learn to *detect and localize multilingual bugs* ***in particular***. Then, to actually learn the task-specific knowledge, xLoc trains this fine-tuning model against the downstream task on the small labeled buggy dataset. These last two design elements together address *Challenge 2*.

To validate our design, we have implemented xLoc for software written in Python and C as primary languages (i.e., *Python-C* software), given the high impact of this language combination [31–34] (e.g.., underlying all major machine-learning frameworks) while the majority of existing multilingual bug-diagnosis tools target Java-C [4, 24, 29, 72]. We also curated a labeled Python-C dataset including 3,770 buggy and 15,884 non-buggy samples, largely manually albeit assisted by a tool we developed for identifying the buggy samples. With this dataset and implementation, we evaluated xLoc for (function-level) detection and (line-level) localization of multilingual bugs. Without an immediate peer tool, we chose as baselines CodeT5 [68] (fine-tuned on the same dataset as for xLoc), a pre-trained Transformer with many successes in enabling various similar tasks before, and GPT-3.5/ChatGPT [54] (in the zero-shot setting) given the acclaimed performance of LLMs with a range of software engineering tasks. Our results show that xLoc achieved promising accuracy for multilingual bug **detection** (94.42% precision, 95.54% recall, and **94.98% F1**), which are *3.73∼13.37% higher than fine-tuned CodeT5 and 0.40∼327.28% higher than ChatGPT*. For the more challenging task of multilingual fault **localization**, xLoc exhibited even greater advantages (**87.24%@Top-1** and 91.09%@Top-3/5 accuracy), *4.41∼24.59% higher than fine-tuned CodeT5 and 511.75∼3965.42% higher than ChatGPT*. We further conducted ablation studies which demonstrated the significant contribution of each of the three design elements of xLoc.

Our results highlighted the peculiar merits of xLoc over the two competitive baselines in that the advantages were much greater in more challenging situations (e.g., localization over detection and @Top-1 over @Top-3/5 localization). Moreover, the overwhelming superiority over ChatGPT revealed that for challenging tasks like multilingual bug detection/localization, *general* LLMs may not be ready yet and developing a *task-specific* DL model remains a meritorious option.

To the best of our knowledge, xLoc is the *first DL-based technique/tool for multilingual bug detection and localization*, as well as the first practical tool for locating multilingual bugs in Python-C software without relying on any test case or other execution information. The challenges identified for multilingual bug detection/localization (especially when using DL approaches) and the proposed novel designs for overcoming those challenges can be potentially applied to language combinations beyond Python-C, for which *our open-source implementation* of xLoc may facilitate. The curated

Python-C *buggy dataset* may also benefit other researchers for conducting future multilingual bug studies and developing multilingual fault diagnosis tools.

2 BACKGROUND AND MOTIVATION

We describe background on multilingual bugs, and then our motivating study and design insights.

2.1 Multilingual Development and Multilingual Bugs

Multilingual development refers to the software practice of developing one software project with two or more programming languages that interact. This practice is ubiquitous and there is an increasing number of multilingual software systems in the real world [13, 25, 34, 45, 59, 62, 63]. This underscores the critical need to ensure the quality of such systems. While this approach often enhances the software development process in different ways [1, 8], such as leveraging the advantages of each language to promote code reuse [49, 64], it also introduces extra complexity to the software. This leads to a higher likelihood of bugs [2].

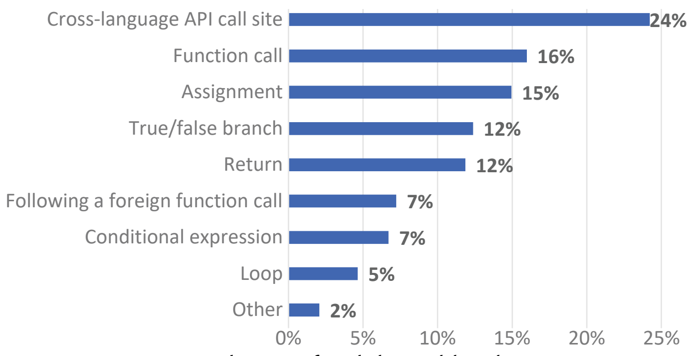


Fig. 1. Distribution of multilingual-bug locations.

In this work, we focus on *multilingual bugs*, which are the bugs that are caused by the interaction of different programming languages. To make two languages (e.g., languages A and B in this example) interact in multilingual software, the code in language A may call a function in language B. This function, usually written by application developers, is a native function. Accordingly, B is considered a native language. Typically, language A may also provide functions that enable code in language B to retrieve information about code in language A. These functions, usually written by A's language developer in A, are foreign functions. Accordingly, A is considered the foreign language here. The APIs designed for two languages to interact, including native and foreign functions, are together referred to as cross-language APIs or cross-language functions.

2.2 Characteristics of Multilingual Bugs

To find the prevalence and the characteristics of multilingual bugs, we did a motivating study on popular open-source projects on GitHub. We collected 2,631 multi-language projects written in C and Python. We collected a set of cross-language functions based on the project documentation and filtered the commits where the changed code has control flow relationship with (reachability to/from) these cross-language functions, named *inter-language commits*. We further filtered the commits with their issue types. If the issue type is relevant to bugs, the commit corresponds to one or more multilingual bugs. For the details of data collection process, please refer to Section 4.1.1.

Table 1 provides the numbers of multilingual bugs in the top-5 projects that have the most stars. Despite these projects' popularity and presumably rigorous development processes, each of these projects has encountered multilingual bugs. The numbers range from as high as 162 bugs in NumPy to 14 bugs in Cython. The presence of multilingual bugs in such esteemed projects underscores the pervasive nature of such bugs. The data serves as a testament to the imperative need for enhanced tools and methodologies to address multilingual bugs.

We randomly sampled 200 real-world multilingual-bug-fixing commits. Then, we conducted comprehensive manual analyses of these commits and derived novel insights into the code locations of prevalent multilingual bugs in practical scenarios.

Table 1. #multilingual bugs in popular projects

	#stars	#forks	#bugs
NumPy [23]	24,587	8,577	162
CPython [16]	55,856	27,722	134
kitty [19]	19,925	862	48
SciPy [66]	11,727	4,855	33
Cython [9]	8,388	1,461	14

As shown in Figure 1, our analysis revealed that many of these multilingual bug locations exhibit characteristics that reflect **various control flow structures** (e.g., loop, return, and true/false branch). This pattern suggests that control flow plays a pivotal role in the introduction of these bugs. Note that these locations are where bugs appeared/manifested, attending to which aligns with our bug-localization task—fault localization typically identifies bug-appearing locations, rather than root causes. For example, the "Return" category refers to bugs occurring at return sites, either in C (including native functions) or Python—yet, in both cases, on cross-language information-flow paths. Cross-language interfacing/interactions are carried via such paths, which include, but not consist solely of, cross-language API calls—(return sites in) pure C or Python functions invoking and/or invoked by those APIs can also be included. Given these findings, and the evident importance of control flow in multilingual code semantics, we are prompted to advocate for the integration of control-flow-based position encoding in deep learning-based models for bug hunting. Such an approach potentially enhances the model's ability to detect and localize multilingual bugs rooted in control flow discrepancies.

Characteristic 1: control flow plays a pivotal role in introducing multilingual bugs.

Besides, we notice that the multilingual bugs were mostly introduced at cross-language API call sites. Among the various sources of multilingual bugs, the cross-language API call site stands out as the most significant. Our analysis indicates that nearly a quarter of all multilingual bugs can be traced back to this specific location. Indeed, they are usually the root causes of the multilingual bugs since they are the interaction sites of different languages. This observation underscores a pronounced correlation between multilingual bugs and the cross-language API call sites. Such a strong association suggests that particular attention should be directed towards these call sites when addressing multilingual bugs.

Characteristic 2: cross-language API callsites tend to be primary multilingual-bug locations.

Finally, we notice that function calls (beyond the calls to cross-language APIs) and assignments (which are also on cross-language information flow paths) emerge as two pivotal bug-introducing locations in multilingual software. Therefore, when analyzing multilingual bugs, these statements should be paid extra attention as well.

Characteristic 3: (non-cross-language) function callsites and assignment statements are also major multilingual-bug locations.

2.3 Design Motivation and Insights

Based on the prevalence of multilingual bugs and their big potential impacts on multilingual software quality, precisely detecting and locating multilingual bugs is crucial. Therefore, we develop a novel approach for this task. Given the three characteristics of multilingual bugs above, we hope that our multilingual bug detection and localization technique can **(1) incorporate control flow relationships between statements; (2) trace the potential bugs back to the causal locations which are mostly the cross-language API callsites; (3) be sensitive to the other function callsites and assignment statements which are also key multilingual bug locations.**

To satisfy the three requirements above, we did a literature review on the existing code analysis techniques that may be used for multilingual bug detection and localization. Existing popular code analysis techniques include dynamic analysis [5, 30, 72], static analysis [29, 55], and learning-based analysis [36]. However, dynamic analysis is limited to finding bugs that can be triggered by test inputs which are either provided with the testing data or generated by fuzzing tools. Based on our literature review, few of the multilingual bug datasets provide test inputs, while fuzzing is uncertain and subject to randomness which may take long time but cannot find bugs. In contrast, static analysis is impeded by disparate semantics of heterogeneous languages. **In this case, we choose to develop a learning-based approach just based on the code to overcome the challenges of building test inputs and manual engineering effort on detection/localization rules.**

To satisfy the requirement of incorporating control flow relationships between statements and tracing the potential bugs back to the root causes which are usually cross-language API calls, we hope that our learning-based technique can incorporate the control flow information as well as build connection between the root causes, which are usually the cross-language API calls, and the statements that have control flow relationships to the root causes. Therefore, we did a literature review on learning-based software engineering (SE) techniques. Recently, deep learning (DL) based models are widely leveraged for SE tasks and they achieve big successes [43]. Popular DL models for SE include recurrent neural network (RNN) [47], graph neural network (GNN) [79], convolutional neural network (CNN) [40], and Transformer [65]. Among them, Transformer uses the self-attention mechanism to learn the attentions between different elements (e.g., code tokens) in the model inputs [65]. There are different designs for the self-attention mechanism. The original Transformer uses the encoding of the absolute position of the elements to bias the attention [65]. Shaw et al. [60] propose relative position encoding which uses the difference between token indexes to bias the attention. To make Transformer work better on programming languages, Peng et al. [57] propose TPTrans which integrates the paths on the abstract syntax tree (AST) of the code to bias the attention. **Given the flexibility of the Transformer attention mechanism design which allows the technique to incorporate the control flow information and build attention between cross-language API calls and statements that have control flow relationships to the control flow API calls, we choose to develop our multilingual bug detection and localization technique with the Transformer architecture**.

To satisfy the requirement of being sensitive to the function call and assignment statements, we hope that the Transformer model understands the code syntactic and semantic information because function call and assignment statements are important syntactic structures and involve important semantic information. However, making DL models understand syntax and semantics of different programming languages requires a large amount of training data. Yet, existing multilingual bug training data we can collect (<20K) cannot satisfy the need of data. Luckily, Transformer can be pre-trained on general tasks with a large amount of corpus available in the wild before training on the specific tasks. This is proved to be effective on different domains of tasks (e.g., natural language processing [12], code understanding [50], code generation [61]). **Therefore, we choose to pre-train our Transformer-based bug detection and localization model to make it sensitive to the function call and assignment statements.**

Based on the choices above, we designed xLoc, a DL-based multilingual bug detection and localization technique based on a pre-trained Transformer model. xLoc is pre-trained on **five pre-training objectives** which enable the model to understand multilingual code syntactic and semantic information and be sensitive to the function call and assignment statements. During the pre-training, xLoc incorporates the control flow information by a **control-flow-based position encoding**. To make the model be able to trace the potential bugs back to the root causes which are usually cross-language API calls, xLoc fine-tunes the model with **cross-language-API-aware**

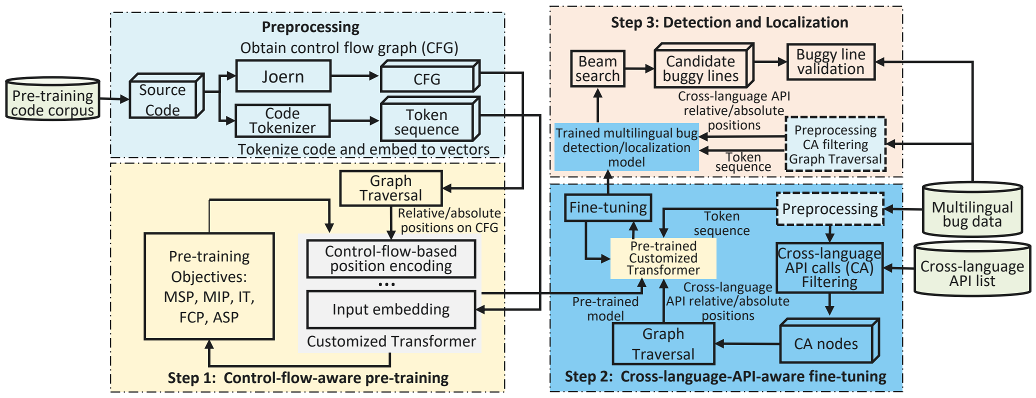

Preprocessing
Obtain control flow graph (CFG)
Pre-training code corpus
Source Code
Joern
CFG
Code Tokenizer
Token sequence
Tokenize code and embed to vectors
Graph Traversal
Relative/absolute positions on CFG
Pre-training Objectives: MSP, MIP, IT, FCP, ASP
Control-flow-based position encoding
Input embedding
Customized Transformer
Step 1: Control-flow-aware pre-training
Step 3: Detection and Localization
Beam search
Candidate buggy lines
Buggy line validation
Cross-language API relative/absolute positions
Trained multilingual bug detection/localization model
Preprocessing CA filtering Graph Traversal
Token sequence
Fine-tuning
Preprocessing
Multilingual bug data
Pre-trained Customized Transformer
Cross-language API calls (CA) Filtering
Cross-language API list
Pre-trained model
Cross-language API relative/absolute positions
Graph Traversal
CA nodes
Step 2: Cross-language-API-aware fine-tuning

Fig. 2. Overview of our multilingual bug detection and localization technique xLoc.

position encoding. For our control-flow-based position encoding and cross-language-API-aware position encoding, we leverage both **absolute position encoding** and **relative position encoding** based on control flow graphs. The absolute position encoding helps locate relevant control-flow constructs within the entire function and differentiate among different functions—hence **helping with (function-level bug) detection**. The relative position encoding helps differentiate among statements/tokens within the function, which **helps with (line-level bug) localization**. By enhancing both the position encoding schemes, xLoc pays greater attention to relevant control-flow constructs in multilingual code towards helping detect and locate multilingual bugs. Which kinds of bugs xLoc can detect/localize is intuitively affected by those covered in the fine-tuning dataset—the one we used includes both crash and semantic bugs. xLoc can detect/localize both kinds, and other kinds it is fine-tuned against.

3 TECHNICAL APPROACH

We start with an overview of xLoc's design and then describe the details of each module and step.

3.1 Overview

Figure 2 shows the overview of xLoc. With the design motivation and insights above, our designed xLoc achieves the goal of multilingual bug localization with three step: (1) control-flow-aware pre-training, (2) cross-language-API-aware fine-tuning, and (3) detection and localization. Each step also comes with a preprocessing module which preprocesses the code samples into the formats that each step can utilize.

In the module preprocessing, xLoc tokenizes the source code into a sequence of tokens, as the Transformer model processes the data as a sequence. Since xLoc incorporates the multilingual-specific information with control flow information, the preprocessing module also parses the source code into control flow graphs (CFGs) for each step to utilize. The token sequences and the parsed CFGs are further utilized in each of the three main steps.

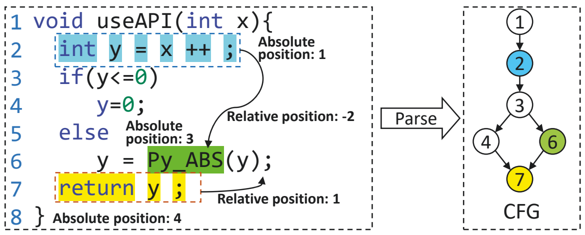
In ***Step 1: control-flow-aware pre-training***, xLoc learns to understand the code syntax and semantics of multiple programming languages by pre-training the Transformer model on a large scale code corpus. It first preprocesses the pre-training code corpus which is in a large scale (>100k pre-training samples). Then, it traverses the parsed CFGs from the preprocessing module and obtains the CFG-based absolute and relative positions for each token or each pair of tokens. The CFG-based absolute position of a token indicates the control flow position of the statement that the token is in. The CFG-based relative position of a pair of tokens indicates the control flow distance of the two statements that two tokens are in. Then, CFG-based absolute and relative positions, along with the token sequences, are fed into a customized Transformer which has a specific-designed control-flow-based position encoding. This makes the model control-flow aware, which is crucial for multilingual bug detection and localization. With the processed pre-training code corpus, xLoc pre-trains the customized Transformer with five pre-training objectives: masked span prediction (MSP), masked identifier prediction (MIP), identifier tagging (IT), function call statement prediction (FCP), and assignment statement prediction (ASP). The five pre-training objectives makes the Transformer model understand the code syntactic and semantic information and be sensitive to the function call and assignment statements, which significantly reduces the needs of large amounts of training data for multilingual detection and localization.

In ***Step 2: cross-language-API-aware fine-tuning***, xLoc fine-tunes the pre-trained model which can understand the code syntax and semantic of multiple programming languages from Step 1 for multilingual bug detection and localization. It first preprocesses the existing multilingual bug training data into token sequences and CFGs. To make the Transformer model focus on the cross-language API calls which are usually the root causes of the multilingual bugs, and the statements that have close control flow relationships to the cross-language API calls, xLoc filters the set of CFGs and only reserve the CFG nodes that involve cross-language API calls (CAs), named CA nodes. Then, it traverses the CFG with only CA nodes remaining to get a set of cross-language API relative and absolute positions. The cross-language API positions, along with the token sequences, are fed into the customized pre-trained Transformer for multilingual bug detection and localization fine-tuning. This makes the model more focused on the cross-language APIs which are usually the root causes of multilingual bugs. After the fine-tuning, the customized Transformer is expected to detect and locate multilingual bugs given new programs.

In ***Step 3: detection and localization***, xLoc takes the fine-tuned model from Step 2 for multilingual bug detection and localization. Similar to Step 2, the multilingual bug testing data is preprocessed and filtered CA nodes. Then, the fine-tuned model takes the token sequences and cross-language API positions to detect and locate the possible multilingual bugs in the given program. To better evaluate the performance of localization for practical use, we leverage beam search which allows the model to output multiple candidate buggy lines, as developers may want to check multiple predictions to find the really buggy one. We evaluate the effectiveness of xLoc by validating the predicted buggy lines with the ground truths.

3.2 Preprocessing

To enable the control-flow-aware pre-training and cross-language-API-aware fine-tuning in the customized Transformer model, we first preprocess the source code into a sequence of tokens and CFGs so that the model can accept them. Among tools that can parse the code and construct its CFG, we choose Joern [73] because: (1) it can parse the code and get the control flow graphs without compiling the whole projects, which saves much time and storage for us; (2) Joern supports parsing different programming languages which allows us to do multilingual code analysis.



```
void useAPI(int x){
    int y = x ++ ;
    if(y<=0)
        y=0;
    else
        y = Py_ABS(y);
    return y ;
}
```

Fig. 3. Example cross-language API call control flow.

Given a code sample, its CFG is a multi-edge graph $g(V, E)$, where V is a set of nodes which represent the statements in the sample, and E is a set the edges connecting the nodes in V, which represent the control flow relationship between different statements. Particularly, the graph starts from the program entry, branches at the conditional statements (e.g., if, for, while, switch) and terminates at the program exit. Figure 3 shows an example. The code is parsed into its CFG using

Joern. The number in each CFG node indicates the line in the source code. We can see that because of the if-else statement, the CFG has branches for lines 4 and 6.

To tokenize the source code into token sequences, we choose to use tree-sitter [41] because: (1) it supports multiple programming languages; (2) it is fast and easy to set up; (3) it does not need input code to be compilable. After the tokenization, the source code is parsed into sequences of tokens that can be accepted by the Transformer input embedding layer.

3.3 Control-Flow-Aware Pre-Training

In this step, xLoc learns to understand the code syntax and semantics of multiple programming languages by pre-training our customized Transformer that is control-flow-aware, with a large scale code corpus (>100k pre-training samples). We separately describe the graph traversal, customized Transformer, and pre-training objectives in details:

3.3.1 Graph Traversal. To make the customized Transformer control-flow-aware, we first traverse the CFGs parsed from the preprocessing module to get the absolute and relative positions on CFG. To get the absolute position on CFG, we traverse each single CFG node on the graph and compute the shortest path to the root node of the CFG (the definition line of the function) with the breadth-first search (BFS) algorithm. The length of the shortest path to the root node is the absolute position of the node. To get the relative position on CFG, we traverse each pair of nodes on the CFG and compute the shortest path between them. The length of the shortest path is the relative position of the pair of nodes. Figure 3 shows an example of the absolute and relative positions on the CFG. Take line 6 as an example, the shortest path to the root node of the CFG is 3, thus its absolute position is 3. The pair of line 2 and line 6 has the shortest path with 2 steps, thus the relative position of the pair line 2 and line 6 is -2 (because line 2 is before line 6; in contrast, the relative position of the pair line 6 and line 2 is +2).

Then, the relative/absolute position information on the CFG, along with the token sequences from the preprocessing module, is fed into the customized Transformer.

3.3.2 Customized Transformer. In a typical Transformer model with traditional absolute and relative position encoding, self-attention is one of the core components. In short, the self-attention module uses *key* and *value* pairs to *query* a dictionary which can be computed as follows:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d}})V \tag{1}$$

where Q, K, V are the previous layer hidden representations of query, key, and value respectively and d is the dimension of the representations. For simplicity, we rewrite and expand the equation to better explain it:

$$z_i = \sum_{j=1}^{n} \frac{exp(a_{ij})}{\sum_{j'=1}^{n} exp(a_{ij'})}(x_j W^V + r_{ij}^V) \tag{2}$$

where $a_i j$ is the attention between the i-th token and the j-th token and can be further expanded as

$$a_{ij} = \frac{1}{\sqrt{2d}}(x_i W^Q)(x_j W^K + r_{ij}^K)^T + \frac{1}{\sqrt{2d}}(a_i^Q)(a_j^K)^T \tag{3}$$

In Equation (2) and (3), we have added the traditional absolute position encoding which are represented as a_i^Q and a_j^K as well as the traditional relative position encoding which are represented as r_{ij}^V and r_{ij}^K. The traditional absolute encoding and relative position encoding are based on the indexes of the tokens in the input sequence. Specifically, absolute position encoding has $a_i^Q = p_i A^Q$

and $a_j^K = p_j A^K$ where p_i and p_j are the sine and cosine functions of the positions i and j and A^Q and A^K are the weight matrices of the encoding [65]. The relative position encoding has $r_{ij}^V = (i-j)R^V$ and $r_{ij}^K = (i-j)R^K$ where R^V and R^K are the weight matrices of the encoding [68].

Since the traditional position encoding directly uses token indexes to compute the encoding, the attention layer may not be able to capture the syntactic and semantic position relationship between the tokens in the code. In Section 3.1, we discussed that control flow is a good representation for code syntactic and semantic information. Thus, we add our control-flow-based absolute and relative position encoding to the model.

$$z_i = \sum_{j=1}^{n} \frac{exp(a_{ij})}{\sum_{j'=1}^{n} exp(a_{ij'})} (x_j W^V + r_{ij}^V + r_{ij}^{V_{CFG}}) \tag{4}$$

$$a_{ij} = \frac{1}{\sqrt{2d}} (x_i W^Q)(x_j W^K + r_{ij}^K + r_{ij}^{K_{CFG}})^T + \frac{1}{\sqrt{2d}} (a_i^Q)(a_j^K)^T + \frac{1}{\sqrt{2d}} (a_i^{Q_{CFG}})(a_j^{K_{CFG}})^T \tag{5}$$

where $r_{ij}^{V_{CFG}}$ and $r_{ij}^{K_{CFG}}$ are the control-flow-based relative position encoding, and $a_i^{Q_{CFG}}$ and $a_j^{K_{CFG}}$ are the control-flow-based absolute position encoding. We define them as follows:

$$r_{ij}^{V_{CFG}} = ctrl\_dist(i,j) R^{V_{CFG}} \tag{6}$$

$$r_{ij}^{K_{CFG}} = ctrl\_dist(i,j) R^{K_{CFG}} \tag{7}$$

$$a_i^{Q_{CFG}} = ctrl\_dist(i,0) A^{Q_{CFG}} \tag{8}$$

$$a_j^{K_{CFG}} = ctrl\_dist(j,0) A^{K_{CFG}} \tag{9}$$

where $R^{V_{CFG}}$ and $R^{K_{CFG}}$ are the weight matrices of the control-flow-based relative position encoding. $A^{Q_{CFG}}$ and $A^{K_{CFG}}$ are the weight matrices of the control-flow-based absolute position encoding. $ctrl\_dist(i,j)$ is the control flow distance between two statements that enclose the i-th and j-th tokens. Figure 3 shows an example. All the tokens in line 2 (in cyan) have absolute position $ctrl\_dist(i,0) = 1$ and relative position $ctrl\_dist(i,j) = -2$ to the token `Py_ABS` (in green) at line 6.

3.3.3 *Pre-Training Objectives.*

With the customized Transformer which is control-flow-aware, we perform pre-training before fine-tuning to gain the model awareness of the syntax and semantics of different programming languages. We use three existing general-purpose programming language-oriented objectives from CodeT5 [68]: `MSP`, `IT`, and `MIP`; and two new objectives that are relevant to multilingual bug detection and localization: `FCP` and `ASP`. We separately describe the five pre-training objectives below.

CodeT5 Objectives. To make the model learn general code syntax and semantics for different programming languages comprehensively, CodeT5 utilizes three pre-training objectives. The first one is `mask span prediction (MSP)`, where the code tokens in an input are randomly masked for the model to recover. The second objective is `identifier tagging (IT)`, where the model learns to tag the tokens in the code that are identifiers. The third objective is `masked identifier prediction (MIP)`, where the model learns to recover the identifiers in the code based on the code semantics. For more details, please refer to the original CodeT5 paper [68].

Function Call Statement Prediction (FCP). Based on our motivating study in Section 2, we found that the multilingual bugs mostly happened in function call statements. Therefore, we hope that the model pays more attention to the function call statements. Therefore, we introduce the `function call statement identification (FCP)` pre-training objective. In this objective, we randomly mask a statement that involves a function call. Then, the model learns to recover the masked statement based on the context semantics of the code.

Assignment Statement Prediction (ASP). In our motivating study, we also notice that multilingual bugs often happen in assignment statements. Thus, similar to FCP, we also pre-train the model for *assignment statement prediction (ASP)*. Again, we randomly mask a statement that involves a variable assignment. Then, the model learns to recover the masked statement based on the context semantics of the code.

To effectively pre-train the model, we first load the pre-trained CodeT5 model which is available on HuggingFace [71]. We then pre-train ASP and finally FCP as multilingual bugs are more likely to happen in function calls (see Figure 1). After the pre-training, the model is used for the cross-language-API-aware fine-tuning.

3.4 Cross-Language-API-Aware Fine-Tuning.

In this step, xLoc fine-tunes the pre-trained model from step 1 for multilingual bug detection and localization. Similar to step 1, the multilingual bug detection and localization data used for fine-tuning is preprocessed to get the CFGs and the token sequences. Based on our motivating study in Section 2, we notice that most of the multilingual bugs happen at the cross-language API calls or the statements which have close control flow relationships to them. Thus, we hope that the model focuses more attention on these statements rather than all the statements on the CFGs. Therefore, for fine-tuning, given a code sample, we filter the nodes in the CFG and only get the CFG nodes that involve cross-language API calls (CA nodes). Then, we traverse the CFG like what we do in step 1, but only the CA nodes and the nodes that have direct control flow relationship to the CA nodes are used for control-flow-based position encoding.

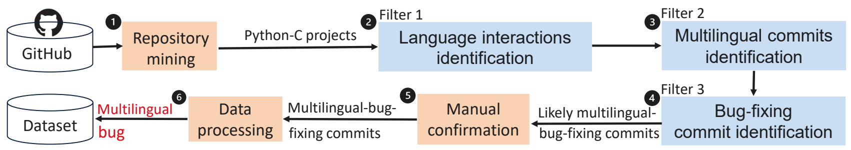
Specifically, for control-flow-based absolute position encoding, only the CFG nodes/statements involving cross-language API calls (i.e., CA nodes) as well as the nodes/statements between the CA nodes and the root node are used for the encoding. For example, in Figure 3, line 6 is used for the absolute position encoding because it has a cross-language API call `Py_ABS`. Line 2 is also used for the absolute position encoding because it is on the path of line 6 to the root of the CFG. However, line 7 is not used because it is neither of the two conditions above. For relative position encoding, only the pairs of statements where at least one of them involving cross-language API calls are used for the encoding. For example, the pair of line 2 and line 6 and the pair of line 6 and line 7 are used for the relative position encoding because they involve a cross-language API call `Py_ABS`, but the pair line 2 and line 7 is not used because the pair does not involve cross-language API call. This removes the control-flow attention on those statements not relevant to cross-language API calls, making the model more focused on the parts that are most likely to have multilingual bugs.

For fine-tuning, we collect a dataset for multilingual bug detection and localization $\mathcal{D} = \{(f_i, s_i)\}$ where each sample is a buggy or non-buggy function f_i and its prediction ground truth s_i. If the function is buggy, s_i is the ground-truth buggy line. If the function is non-buggy, s_i is a token 0. Both f_i and s_i are in text form to fit the Transformer. To fine-tune the model with parameters θ, we minimize the following negative log-likelihood loss to maximize the likelihood estimation:

$$L(\mathcal{D};\theta) = \sum_{i=1}^{N} -log\ p(s_i|f_i;\theta) \tag{10}$$

3.5 Detection and Localization

After the fine-tuning, xLoc is expected to detect the multilingual bugs and output the buggy line if the given function is buggy. Given a code sample, xLoc preprocesses the code sample, filters CA nodes, and traverses the CFG to get the cross-language API relative/absolute positions and the token sequence like what it does in step 2. Then, the model predicts whether the code sample is buggy, and if so, which line is the buggy line. Since the model is probabilistic, it allows to predict multiple buggy lines with beam search size greater than 1. When setting the beam search size K,


GitHub
1
Repository mining
Python-C projects
2 Filter 1
Language interactions identification
3 Filter 2
Multilingual commits identification
4 Filter 3
Bug-fixing commit identification
Likely multilingual-bug-fixing commits
5
Manual confirmation
Multilingual-bug-fixing commits
6
Data processing
Multilingual bug
Dataset
Fig. 4. The workflow of our data collection process.

xLoc output K lines that are the most likely ones to be buggy. Later we will use different beam search sizes to evaluate xLoc and the baseline techniques.

4 EVALUATION

In this section, we evaluate the effectiveness of xLoc for detecting and localizing multilingual bugs. We seek to answer the following research questions:

- **RQ1:** How effective is xLoc in multilingual bug detection compared to other approaches?
- **RQ2:** How effective is xLoc in multilingual bug localization compared to other approaches?
- **RQ3:** How much does each of the design elements of xLoc contribute to its performance?

We first describe the evaluation dataset, metrics, and the setup. Then, we answer these questions.

4.1 Experimental Design

4.1.1 Dataset. We utilize two distinct datasets; one is employed for the pre-training of the model, while the other dataset is used for the model's fine-tuning and evaluation.

Pre-training. We build the pre-training datasets by using the C and Python programs from IBM CodeNet [58]. We select 137,864 C programs and 171,358 Python programs. Since the file-level programs may be too long for the model to learn, we extract the functions in these programs and obtain 385,447 samples. These function-level samples are the actual data for pre-training.

Fine-tuning & evaluation. Due to the absence of multilingual bug datasets, collecting such datasets is paramount for model training. We collect data from GitHub, given its prominence in software development and its vast repository of real-world code and metadata. We choose to focus on Python and C, because of two reasons: (1) both Python and C are top-10 languages with extensive developer communities; (2) C is renowned for its high performance and efficient memory management, while Python is celebrated for its simplicity and readability. Python and C, when combined, have a large ecosystem of libraries and modules that can be used for various tasks, such as machine learning (ML) and artificial intelligence (AI).

Figure 4 shows the workflow of the data collection process. We first scrape 2,631 Python-C repositories from GitHub in descending order based on the number of stars. Then, from these repositories, we extract the multilingual commits which are the commits involving interactions between different languages. To get the multilingual commits, we do the following two steps:

Since not all the projects written in Python and C involve interactions between the two languages, the first step is to identify the multilingual projects with language interactions. To do so, we employ a state machine from PolyFax [32] to detect Python-C interactions: the state machine determines whether a project has language interaction by recognizing the presence of interaction types (e.g., ctypes, Cython, SWIG) in the project. For more details, please refer to the original PolyFax paper [32].

After filtering the projects involving language interactions, the second step is to filter the multilingual commits. We define multilingual commits as the commits whose changed code lines can reach or can be reached by the cross-language API calls on the control flow graph. To do so, we built a tool based on Joern [73], and enhanced it with whole-program control flow reachability analysis, to automate this process.

Then, we filter the multilingual bug-fixing commits from these multilingual commits, because not all the commits are bug fixing. To do so, we get the linked issue posts of the multilingual commits on Github. Each issue has a classification label. We filter the multilingual commits with the issue labels related to bugs. Our bug-related labels include: `'error'`, `'bug'`, `'defect'`, `'patch'`, `'mistake'`, `'fault'`, `'failure'`, `'fix'`, `'issue'`, `'incorrect'`, `'flaw'`. Finally, the resulting commits are still likely to be false positives as multilingual bug-fixing commits. Thus, we followed a manual confirmation process in which each commit was checked to confirm that it is a multilingual bug-fixing commit—which fixes the faulty underlying cross-language information flow. We finally obtained 1,789 potential multilingual-bug-fixing commits from 56 repositories.

Each bug commit is a pair of code which involves the versions before and after the commit. Based on this, we use the diff command provided by Git to get the lines modified and the modified lines are the buggy lines we want to locate. The commits are based on whole source code files and the code in the files is usually very long (e.g., > 5000 tokens). Since the Transformer model is sensitive to the input text length [10, 52], inputting the whole files is not feasible. Thus, we extract the respective functions where the modified lines are in. Then, we only keep the functions that contain cross-language interactions, and thus the buggy functions are the actual inputs to the model. For each buggy sample, the expected output (ground truth) is the buggy line in the function. If there are multiple buggy lines, we use the first buggy line as the ground truth since it is usually the most important one and the rest of the lines may be directly located after it [37]. As for non-buggy functions, we extract the function from the commit without the buggy tag in the previous step. For each non-buggy sample, the expected output (ground truth) is 0.

We finally collect 19,654 samples, including 3,770 buggy samples and 15,884 non-buggy samples in C and Python. We split this dataset 8:1:1 for training (fine-tuning), validation, and testing.

4.1.2 *Metrics.*

Following other prior bug detection/localization studies [36, 37], we use recall, precision, and F1 as the metrics to evaluate the detection effectiveness of xLoc and the baselines. We use top-K as the metric to evaluate the localization effectiveness of xLoc and the baselines. Specifically, when we set the beam search size as K, xLoc outputs K predictions. For detection, we only check the first prediction, no matter what k is: if the prediction is 0, the code sample is predicted as non-buggy, otherwise, it is predicted as buggy and the output should be the first line of the buggy code. If the code sample is predicted as buggy, we check all the k predictions which are most likely to be the buggy lines. Once one of the predictions matches the ground truth, we mark that sample to be a correct case. The percentage of correct cases in all the testing samples is the accuracy at top-K. We report top-1, top-3, and top-5 following the prior works [36, 37].

4.1.3 *Setup.*

We use the pre-trained CodeT5 on HuggingFace [71] as the base and develop xLoc by directly modifying its source code. We use the default hyper-parameters of CodeT5 for training and testing. We perform our experiments on a workstation with an AMD Ryzen Threadripper 3970X (3.7GHz) CPU with 32 Cores, an Nvidia GeForce RTX 3090 GPU, and 256GB memory.

4.1.4 *Baselines.*

We assess the effectiveness of xLoc over two baselines:

- **GPT-3.5 Turbo** is one of the latest Large Language Models (LLMs), which is trained with 175 billion parameters. It is characterized by its ability to process up to 16,385 tokens in a single context. We use this model in the *zero-shot* setting as zero-shot LLMs recently showed very-promising performance for bug diagnosis [56], while it is not clear yet how LLMs can be *better* prompted for multilingual bug detection/localization.
- **CodeT5** [68] is a cutting-edge pre-trained encoder-decoder Transformer model specifically designed for code understanding and generation tasks. With xLoc, we aim to demonstrate how to leverage a pre-trained code language model for multilingual bug detection/localization

Table 2. Effectiveness of xLoc and the baselines for detection and localization.

	GPT-3.5-turbo	CodeT5	xLoc
Recall	94.04% (0.40% ↑)	84.27% (13.37% ↑)	**95.54%**
Precision	22.36% (327.28% ↑)	91.02% (3.73% ↑)	**94.42%**
F1	36.13% (162.88% ↑)	87.51% (8.53% ↑)	**94.98%**
Top1	2.14% (3965.42% ↑)	70.02% (24.59% ↑)	**87.24%**
Top3	4.96% (1736.49% ↑)	81.89% (11.23% ↑)	**91.09%**
Top5	14.89% (511.75% ↑)	87.24% (4.41% ↑)	**91.09%**

through model customization and further pre-training, for which the model should be open-source and architecture-modifiable. CodeT5 is such a model widely adopted in AI4SE [3, 28, 67] (often via fine-tuning). Thus, we chose CodeT5 as a baseline.

In particular, for GPT-3.5 (i.e., ChatGPT), we used two prompt templates both starting with the *objective* and followed by the *output requirements*. The first aims at bug detection:

> Given the following code, tell whether the code is buggy: `[CODE]`
> Output: 'YES' or 'NO'; no explanations.

where `[CODE]` refers to the code in each testing sample. If the response was 'YES', indicating the presence of a bug, we continue to ask the LLM to localize the bug with the second prompt template:

> Given the previous code, identify the top-5 most likely buggy locations.
> Output: each location with the probability it's buggy as {code line, probability}; no explanations.

In the case of positive bug detection, we query the LLM for the top-5 bug locations so that we can compute the model's bug localization performance in terms of the metrics defined earlier (i.e., top-1, top-3, and top-5 accuracy), as for the other baseline CodeT5 and our technique xLoc.

4.2 Results

4.2.1 RQ1: Detection Effectiveness of xLoc. Table 2 delineates the efficacy of xLoc in comparison to baseline methods on 1912 test samples. The recall, precision, and F1 score achieved by xLoc are 95.54%, 94.42%, and 94.98%, respectively. The commendable precision underscores the high quality of the generated samples, while the recall of 95.54% attests to its universality in supporting large-scale bug detection. An F1 score of 94.98% further manifests the overall effectiveness of xLoc.

Columns 2 and 3 of Table 2 present the effectiveness of two baseline methods, with the figures in parentheses indicating the relative improvement of xLoc over these baselines. In terms of recall, precision, and F1 score, xLoc exhibits advantages of 0.40%, 327.28%, and 162.88% over the LLM GPT-3.5, respectively. Despite GPT-3.5 being an expansive and potent large language model, it lacks the requisite fine-tuning for bug detection. Its recall is nearly indistinguishable from xLoc, which we surmise results from its heightened sensitivity to bugs, leading to the misclassification of numerous negative samples as positive. Compared to CodeT5, xLoc surpasses in recall, precision, and F1 score by 13.37%, 3.73%, and 8.53%, respectively. This is attributed to xLoc's utilization of CFG-based pre-training with FCP/ACP objectives and CFG-based fine-tuning.

4.2.2 RQ2: Localization Effectiveness of xLoc. The last three rows of Table 2 depict the capability of each model in terms of localization. The 'top1' metric represents the accuracy of each model when only one localization prediction is permissible. 'Top3' and 'top5' metrics, on the other hand, denote the accuracy when models provide three and five predictions, respectively. With only one prediction, xLoc successfully pinpoints 87.24% of bugs, indicating its prowess in swiftly locating the majority of bugs. When xLoc offers three and five predictions, the accuracy escalates to an impressive 91.09%.

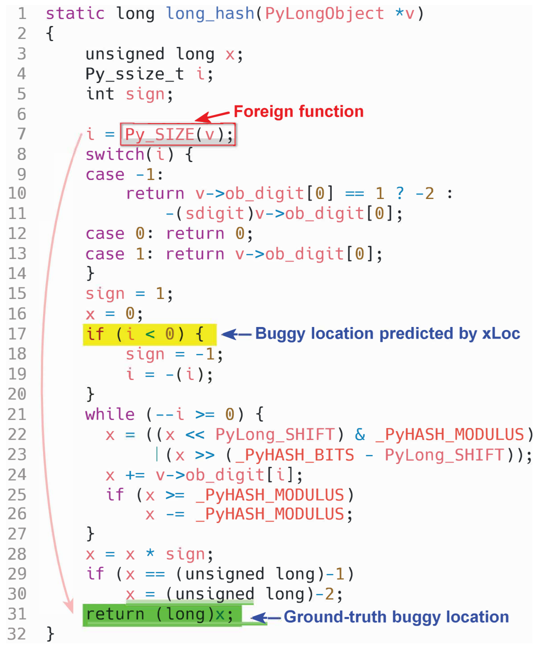
Table 3. Effectiveness of xLoc and the baselines for localization for various bug locations

Bug location	#samples	xLoc			codeT5			GPT-3.5-turbo		
		Top1	Top3	Top5	Top1	Top3	Top5	Top1	Top3	Top5
Cross-Language API Callsite	114	**91.23%**	**96.49%**	**96.49%**	82.46%	83.33%	84.21%	2.63%	6.14%	18.42%
Function Call	58	**86.21%**	**91.38%**	**91.38%**	72.41%	74.14%	74.14%	0.00%	0.00%	6.90%
Assignment	63	**80.95%**	**84.13%**	**84.13%**	76.19%	79.37%	79.37%	0.00%	6.35%	17.46%
True/False Branch	38	**71.05%**	**76.32%**	**76.32%**	60.53%	63.16%	63.16%	2.63%	7.89%	28.95%
Return	20	**65.00%**	**75.00%**	**75.00%**	65.00%	75.00%	75.00%	0.00%	0.00%	5.00%
Following a Foreign Function Call	13	84.62%	**92.31%**	**92.31%**	**92.31%**	92.31%	92.31%	15.38%	15.38%	15.38%
Conditional Expression	12	**75.00%**	**75.00%**	**75.00%**	58.33%	58.33%	58.33%	0.00%	8.33%	33.33%
Loop	2	**50.00%**	**100.00%**	**100.00%**	50.00%	50.00%	50.00%	0.00%	0.00%	50.00%
Other	11	**72.73%**	**81.80%**	**81.82%**	72.73%	72.73%	72.73%	0.00%	0.00%	0.00%

This accentuates xLoc's formidable capacity in bug localization. The consistent accuracy between 'top3' and 'top5' suggests that the top three predictions already furnish substantial reference value.

In bug localization, xLoc exhibits a large superiority over GPT-3.5, leading by staggering margins of 3965.42%, 1736.49%, and 511.75%. xLoc markedly eclipses the performance of GPT-3.5 in this task. One plausible explanation for this disparity is each model's inherent training methodologies, with xLoc potentially leveraging more multilingual-specific insights and techniques tailored for bug localization.

When juxtaposed with CodeT5, xLoc maintains a lead in bug localization by margins of 24.59%, 11.23%, and 4.41%. The advantages of xLoc, particularly in the 'top1' and 'top3', are palpable. Notably, the improvement observed in the 'top1' metric is twice as pronounced as that in the 'top3' metric. This differential is attributed to the fact that CodeT5 lacks the multilingual-specific knowledge that xLoc accrues during its pre-training and fine-tuning phases. Such specialized knowledge, embedded within xLoc, equips it with a more nuanced understanding of multilingual bugs, thereby enhancing its localization capabilities.


```
static long long_hash(PyLongObject *v)
{
    unsigned long x;
    Py_ssize_t i;
    int sign;

    i = Py_SIZE(v);
    switch(i) {
    case -1:
        return v->ob_digit[0] == 1 ? -2 :
            -(sdigit)v->ob_digit[0];
    case 0: return 0;
    case 1: return v->ob_digit[0];
    }
    sign = 1;
    x = 0;
    if (i < 0) {
        sign = -1;
        i = -(i);
    }
    while (--i >= 0) {
      x = ((x << PyLong_SHIFT) & _PyHASH_MODULUS)
           |(x >> (_PyHASH_BITS - PyLong_SHIFT));
      x += v->ob_digit[i];
      if (x >= _PyHASH_MODULUS)
          x -= _PyHASH_MODULUS;
    }
    x = x * sign;
    if (x == (unsigned long)-1)
        x = (unsigned long)-2;
    return (long)x;
}
```

Fig. 5. An example where xLoc failed to locate a bug of the "Return" category (i.e., at a return site).

For a closer look into the localization performance of xLoc against the two baselines, Table 3 gives its effectiveness for each of the possible bug-location categories considered (Figure 1). As shown, xLoc performed the best (as highlighted in boldface) in all cases except for one category, `Following a Foreign Function Call` (which accounts for only 13/331=*4%* of all the positive within-function testing samples), for top-1 accuracy.

Across the nine location categories, in terms of top-1 accuracy only, xLoc performed the worst for bugs of the "Loop" category (with top-1 accuracy of 50%). Meanwhile, *overall*, locating bugs at "Return" sites was the most challenging for xLoc. These worst cases can be attributed to (1) our design focus for xLoc's pre-training and fine-tuning phases and (2) the relationships between the multilingual-bug locations as we categorized.

First, our model is pre-trained to pay more attention to two primary kinds of locations, (cross-language and other) function callsites and assignment, while then fine-tuned to attend more to

Table 4. Description of ablated Versions

	FCP/ASP	API-aware FT	FCP/ASP + API-aware FT	xLoc
pre-training with FCP/ASP	✓	✗	✓	✓
CFG-based position encoding in FCP/ASP	✗	✗	✗	✓
Cross-language-API-aware position encoding in fine-tuning	✗	✓	✓	✓

Table 5. Performance of the ablated versions of xLoc for detection and localization.

	FCP/ASP	API-aware FT	FCP/ASP + API-aware FT	xLoc
Recall	94.65%	91.69%	93.17%	**95.54%**
Precision	92.73%	95.37%	**96.02%**	94.42%
F1	93.68%	93.49%	94.57%	**94.98%**
Top1	84.5%	82.78%	85.16%	**87.24%**
Top3	90.80%	86.35%	88.42%	**91.09%**
Top5	90.80%	88.42%	88.42%	**91.09%**

locations close to cross-language function callsites. Second, a detailed examination of our datasets revealed that other kinds of location (e.g., “true/false branches”, “conditional expression”) often incorporate or are located near those primary focus areas. This explains why xLoc, although not directly pre-trained/fine-tuned on those other kinds of location, still worked reasonably well for those bug locations. However, return sites and loop statements typically do not encapsulate function calls or assignments (at least in our dataset). Also, in all of our failure cases, the actual faulty "Return"/"Loop" locations are neither very close to function calls/assignments nor to cross-language functions. Figure 5 illustrates one of such failure cases, where the bug is actually at line 31 (a return site), yet incorrectly located at line 17 by xLoc. The cross-language (foreign) function callsite is at line 7, significantly distant from the buggy line 31. This considerable distance, given xLoc's pre-training and fine-tuning focus, diminishes xLoc's sensitivity to the bug at the return site.

4.2.3 *RQ3: Ablation Studies.* In this section, we do ablation studies by comparing our full design with three ablated versions shown in Table 4:

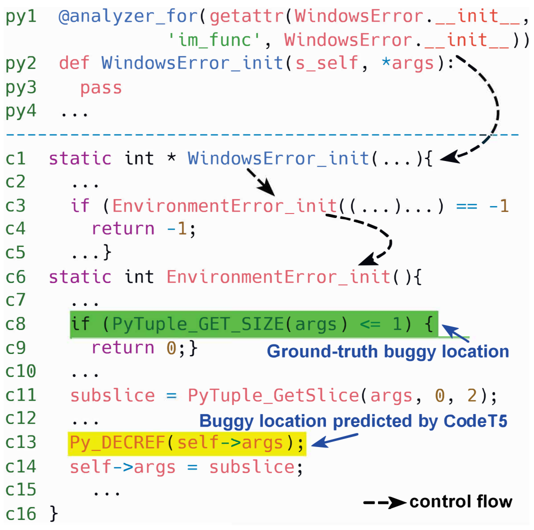
- **FCP/ASP**: We pre-train the FCP/ASP tasks based on CodeT5 without CFG-based position encoding and fine-tune the model without the cross-language-API-aware position encoding.
- **API-aware FT**: We fine-tune the model with cross-language-API-aware position encoding based on CodeT5 without pre-training on FCP/ASP.
- **FCP/ASP + API-aware FT**: We pre-train the FCP/ASP tasks based on CodeT5 without CFG-based position encoding, but we fine-tune the model with cross-language-API-aware position encoding.

As shown in Table 5, the FCP/ASP version shows commendable performance, particularly in terms of recall, where it attains 94.65%, closely tailing the xLoc's leading 95.54%. However, despite its robust pre-training with FCP/ASP, FCP/ASP does not utilize CFG-based position encoding in the pre-training and cross-language-API-aware position encoding in the fine-tuning. This absence is evident in the TopN, where xLoc takes the lead, particularly in the Top1 classification with 87.24% against FCP/ASP's 84.5%. Given xLoc's comprehensive integration of both pre-training techniques - CFG-based positional encoding and FCP/ASP, it demonstrates a more rounded performance.

CFG-based FT stands out for its precision, achieving a significant 95.37% against xLoc's 94.42%. It leverages fine-tuning with CFG-based positional encoding, which likely bolsters its precision in detection. However, in other metrics, particularly the recall and TopN, xLoc exhibits superior performance. The primary distinguishing factor appears to be xLoc's simultaneous use of pre-training with CFG-based positional encoding and FCP/ASP, which CFG-based FT lacks, hinting at this combination's possible advantage.

In terms of performance, the FCI/ASI + CFG-based FT version almost parallels xLoc. It boasts the highest precision of 96.02%, marginally edging out xLoc. This version's utilization of both pre-training with FCP/ASP and fine-tuning with CFG-based positional encoding seems pivotal to its high precision. However, despite these strengths, xLoc still manages to lead slightly in F1 and decisively in the TopN metrics. This performance edge is attributed to xLoc's unique combination of pre-training techniques, which provide it with a more comprehensive multilingual-specific foundation.

In essence, while each ablated version has its strengths, xLoc's holistic combination of pre-training strategies, complemented by fine-tuning with CFG-based positional encoding, facilitates its consistent top-tier performance across the board.



```
py1  @analyzer_for(getattr(WindowsError.__init__,
               'im_func', WindowsError.__init__))
py2  def WindowsError_init(s_self, *args):
py3    pass
py4  ...
---------------------------------------------------
c1   static int * WindowsError_init(...){
c2     ...
c3     if (EnvironmentError_init((...)...) == -1
c4       return -1;
c5     ...}
c6   static int EnvironmentError_init(){
c7     ...
c8     if (PyTuple_GET_SIZE(args) <= 1) {
c9       return 0;}
c10    ...
c11    subslice = PyTuple_GetSlice(args, 0, 2);
c12    ...
c13    Py_DECREF(self->args);
c14    self->args = subslice;
c15      ...
c16  }
```

Ground-truth buggy location
Buggy location predicted by CodeT5
control flow

Fig. 6. An example where xLoc succeeded in locating the bug while CodeT5 did not.

xLoc is also efficient in terms of time cost. It took 2.13 hours (7,666 seconds) in total for bug detection and localization across our testing dataset, hence an average cost of 4 seconds per sample.

5 DISCUSSION

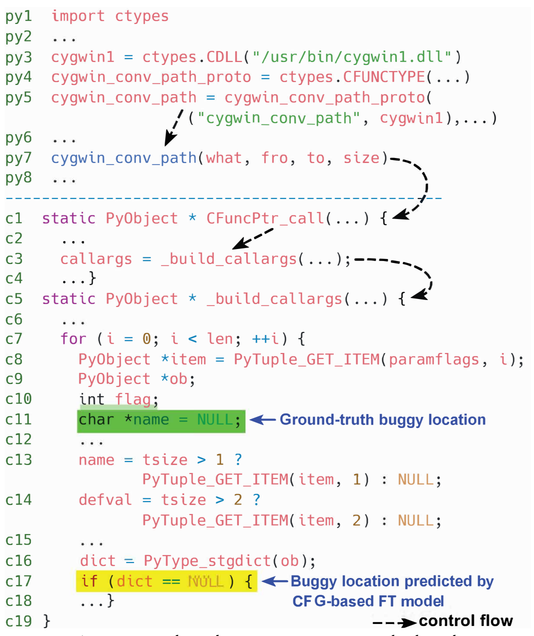
In this section, we showcase a selection of cases that underscore the efficacy of the innovative components integrated into xLoc and delve into the underlying causes for such outcomes. We also provide observations about the advantage of xLoc compared to CodeT5 and GPT-3.5. Finally, we discuss how the methodology xLoc instantiates for Python-C in this paper may be applied to bug detection/localization for other language combinations.

Figure 6 displays an example where xLoc accurately identifies line c8 (marked in green) as the buggy line, while the original CodeT5 points to line c13 (marked in yellow) as the line with the bug. The bug is at line c8 because it is caused by not correctly checking the returned value of PyTuple_GET_SIZE(args). The returned value not only needs to be less than or equal to one but also needs to check the upper bound. xLoc correctly pinpoints line c8 as the multilingual error point by leveraging the control flow relationship between lines py1 and c8 (shown in black dashed line). Since line c8 contains a cross-language API call (PyTuple_GET_SIZE), xLoc can accurately locate this bug line with its heightened sensitivity to cross-language APIs. However, the original CodeT5 lacks CFG-based knowledge, leading to its failure in this instance.

From the first case, xLoc demonstrates a superior capability to accurately pinpoint the exact line of code with a bug, as evidenced by correctly identifying the buggy line in the given example, in contrast to CodeT5. xLoc is adept at recognizing incorrect API usage, and effectively utilizes the control flow relationships between lines of code. It identifies errors even when they are indirectly related to other lines.

Figure 7 presents another example where xLoc accurately identifies line c11 as the buggy line, while the CFG-based FT model points to line c17 incorrectly. The mistake is evident on line c13 due to the incorrect usage of the API, PyTuple_GET_ITEM. The 'char' type variable 'name' is used to store individual characters. However, PyTuple_GET_ITEM typically returns a pointer pointing to PyObject. Attempting to assign a pointer to a 'char' type variable results in an error due to a type mismatch. Moreover, line c13, which tries to assign NULL to a 'char' type, is also problematic. The CFG-based FT model entirely overlooks this bug. Since line c13 invokes the cross-language API, PyTuple_GET_ITEM, xLoc correctly pinpoints line c11 as the multilingual error point by leveraging the control flow relationship between lines c11 and c13. Furthermore, as xLoc is pre-trained on the FCP/ASP objective with CFG-based position encoding, it can pay more attention to assignment statements through the CFG path. Lacking of this pre-training is the reason for the failure of the CFG-based FT model in this instance.

From the second case, we know the model through pre-training on the FCP/ASP objective with CFG-based position encoding can accurately identify and locate multilingual bugs, especially in complex scenarios where other models overlook or misidentify. The model exhibits heightened sensitivity to cross-language API calls, allowing it to pinpoint bugs associated with incorrect API usage more effectively. By utilizing the control flow relationship between lines of code, the model can accurately identify the origins of bugs, even when they manifest in subsequent lines.



```
py1  import ctypes
py2  ...
py3  cygwin1 = ctypes.CDLL("/usr/bin/cygwin1.dll")
py4  cygwin_conv_path_proto = ctypes.CFUNCTYPE(...)
py5  cygwin_conv_path = cygwin_conv_path_proto(
                 ("cygwin_conv_path", cygwin1),...)
py6  ...
py7  cygwin_conv_path(what, fro, to, size)
py8  ...
----------------------------------------------------
c1  static PyObject * CFuncPtr_call(...) {
c2    ...
c3    callargs = _build_callargs(...);
c4    ...}
c5  static PyObject * _build_callargs(...) {
c6    ...
c7    for (i = 0; i < len; ++i) {
c8      PyObject *item = PyTuple_GET_ITEM(paramflags, i);
c9      PyObject *ob;
c10     int flag;
c11     char *name = NULL;
c12     ...
c13     name = tsize > 1 ?
               PyTuple_GET_ITEM(item, 1) : NULL;
c14     defval = tsize > 2 ?
               PyTuple_GET_ITEM(item, 2) : NULL;
c15     ...
c16     dict = PyType_stgdict(ob);
c17     if (dict == NULL ) {
c18     ...}
c19 }
```

Ground-truth buggy location
Buggy location predicted by CFG-based FT model
control flow

Fig. 7. An example where xLoc succeeded in locating the bug while CFG-based FT model did not.

Additional Observations. It was observed that the advantage of xLoc in terms of detection is not significantly greater than CodeT5. However, as for localization, xLoc notably outperforms. While locating the buggy line within the top-5 predictions is difficult, narrowing it down to the exact top-1 is markedly more challenging. Impressively, in this stringent top-1 scenario, xLoc surpasses CodeT5 by a margin of 24.59%. xLoc possesses a more refined capability in precise localization tasks. Such ability can be invaluable in practical applications, potentially reducing the time and effort developers invest in debugging. It further implies that the position encoding of xLoc makes it more adept at discerning the intricacies of multilingual code structures.

While GPT-3.5/ChatGPT is recognized as a relatively mature Language Model, its performance in both detection and localization tasks (at least for multilingual bugs in Python-C software) appeared to be overshadowed by xLoc. Notably, xLoc's proficiency in localization is multiple times superior to GPT-3.5. Presently, while LLMs are experiencing rapid advancements, it is pivotal not to dismiss the value of task-specific models. This suggests that even as general language models continue to evolve and show promise in various scenarios, there remains a distinct advantage in models developed for specific tasks against task-specific datasets. Such specialized models, like xLoc, can learn task-specific knowledge that may not be well captured by general models like GPT-3.5.

Applicability to other language combinations. Our methodology for the task of multilingual bug detection/localization is to (1) pre-train a language model with customized position encoding on objectives specific to this task, followed by (2) fine-tuning the pre-trained model against labeled bug datasets also specific to this task, where the model customization and objective design are immediately based on the characteristics of bug locations obtained from (3) a characterization study. We believe this methodology is applicable to working reasonably well for other language combinations, although the xLoc implementation may not immediately be. To evaluate our approach on another language combination, we would repeat (1)–(3) for it, especially collecting the respective bug dataset which would be the most tedious and costly step. We currently demonstrate our approach for Python-C given the impactful role of Python applications which commonly use native code in C (e.g., popular ML and scientific-computing frameworks like NumPy, PyTorch, TensorFlow, and even the Python language-runtime's standard implementation CPython itself). Also, while we do not have full bug-location distribution data for other language combinations, cross-language function callsites are the primary bug locations for some other combinations too (e.g., Java-C [21]).

6 THREATS TO VALIDITY

Threats to internal validity. The code of our tools and models is a critical component in ensuring the validity of our evaluation results. Despite our efforts to avoid bugs and errors through code reviews, there is still a possibility that undetected errors could exist in the code. These errors might have affected the performance of the tools and models and led to incorrect results or conclusions.

Another internal validity threat is that the case studies in the motivating study on the characteristics of multilingual bugs may have resulted in inaccurate findings/conclusions due to manual biases/errors. To mitigate this threat, three of the authors followed an inter-rater agreement procedure to resolve any dissents, which help reduce human biases via cross checking. Specifically, each of these authors independently analyzed each of the bug cases. After the individual analyses, the three authors compared their initial results in a comprehensive review session. The degree of agreement among the authors was quantified in terms of kappa scores, which were 0.83, 0.84, and 0.79 between the first and the second, the first and the third, and the second and the third authors, respectively. These scores reflect a substantial level of agreement among the raters, underscoring a high degree of consistency in the assessment process, thereby reinforcing the credibility and validity of the evaluation outcomes. Through thorough deliberation and exchange of perspectives, the authors were able to reconcile their differences and ultimately reach a unanimous agreement on the presented study findings.

Threats to external validity. The main external validity threat is that the Python and C code corpus used for pre-training our xLoc model may not be representative of all real-world code in these two languages. To mitigate this threat, we expressly used as much pre-training data for each language as our computing capacities permit. Similarly, the task-specific dataset we used for fine-tuning the pre-trained model may not represent all real-world multilingual bugs in arbitrary Python-C software. Indeed, our ultimate bug dataset is relatively small. Thus, our results should be best interpreted with respect to the bugs represented by the dataset.

On a similar note, the process of curating this bug dataset is subject to human biases/errors since the bug labels were confirmed manually at the last step of the process. To reduce these biases/errors throughout the labeling process, three of the authors performed cross validation by following a similar inter-rater agreement procedure to that applied during the motivating case studies.

7 RELATED WORK

Software fault detection and localization. Existing approaches in this line are either mainly based on program analysis or largely data-driven (e.g., based on machine/deep learning (DL)). The first category predominantly focuses on JNI (i.e., Java-C) programs as one particular case of multilingual software [4, 5, 24, 29, 72], with fewer ones targeting Python-C software [30, 35] which are dynamic techniques. The second category is represented on emerging DL-based fault localization techniques [18, 36–38]. In comparison with most of these existing techniques, which cannot work with multilingual (e.g., Python-C) code as the only input (e.g., they require test cases), xLoc is a Transformer-based Python-C bug detector and localizer that only needs source code as the single input. By nature/definition, spectrum-based and mutation-based fault localization techniques require test cases as well.

Although the technique in [39] does not require test cases, it needs to construct CFG and data flow graph via heavy language-specific static analyses, which immediately impedes their application to multilingual code. Likewise, information-retrieval (IR) based fault localization does not require test cases either; yet it needs quality bug reports that also are often not available to multilingual systems in the wild—and it typically only locates a buggy file, not a function or code line.

Multilingual code analysis and testing. The mutation testing tool described in [22] facilitates mutant generation across different languages via regular-expression-based transformations. On the other hand, Amleto[15] and Gillian[17] are testing platforms that operate by translating VHDL, SystemC, and various languages into a common intermediate representation (IR). Commonly, none of these tools are designed to handle programs composed of multiple language units simultaneously.

On a similar note, FANS [42] specializes in testing Android native system services, predominantly written in C++, which may invoke Java code. However, despite its capability to indirectly trigger Java exceptions, FANS itself functions primarily as a single-language (C++) testing tool. Likewise, Favocado [14] targets the testing of JavaScript engines, with a primary focus on their binding layer, which is written in one language. On the other hand, techniques like ORBS [7] promise language-agnostic analysis capabilities, which may not be practical to use due to scalability barriers [74].

8 CONCLUSION

A defining aspect of multilingual code lies in the interactions/interoperations among the different programming languages used, which are commonly realized through explicit language interfacing via cross-language API calls. Yet multilingual bugs buried at and across these language boundaries constitute a critical challenge to the quality of multi-language software. Existing relevant techniques/tools fall short of identifying these bugs due to several fundamental barriers. We thus presented xLoc, a novel DL-based approach to (function-level) detection and (line-level) localization of multilingual bugs that address those barriers. Our Transformer-based design in xLoc features *control-flow-based position encoding and pre-training* towards learning *general knowledge* about different control structures in multilingual code, and *cross-language-API-vicinity-based position encoding and fine-tuning* towards learning *downstream-task-specific knowledge* for differentiating buggy multilingual functions/statements from non-buggy ones. For demonstrating and validating our design, we implemented xLoc for Python-C software and evaluated it against codeT5 and ChatGPT as baselines on a new multilingual buggy dataset for this language combination. Our evaluation results have shown substantial merits of xLoc over these state-of-the-art baselines for both detection and localization of multilingual bugs.

9 DATA AVAILABILITY

We released all of our code and datasets to facilitate reproduction, replication and reuse.

ACKNOWLEDGMENTS

We thank our associate editor and reviewers for insightful and constructive comments. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CCF-2146233 and in part by the U.S. Office of Naval Research (ONR) under Grant N000142212111.

REFERENCES

[1] Mouna Abidi, Manel Grichi, and Foutse Khomh. 2019. Behind the scenes: developers' perception of multi-language practices. In *Annual International Conference on Computer Science and Software Engineering*. 72–81. https://doi.org/10.5555/3370272.3370280

[2] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are multi-language design smells fault-prone? an empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–56. https://doi.org/10.1145/3432690

[3] Mayank Agarwal, Yikang Shen, Bailin Wang, Yoon Kim, and Jie Chen. 2024. Structured code representations enable data-efficient adaptation of code language models. *arXiv preprint arXiv:2401.10716* (2024), 1–18. https://doi.org/10.48550/arXiv.2401.10716

[4] Sora Bae, Sungho Lee, and Sukyoung Ryu. 2019. Towards understanding and reasoning about Android interoperations. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 223–233. https://doi.org/10.1109/ICSE.2019.00038

[5] Junyang Bai, Weiping Wang, Yan Qin, Shigeng Zhang, Jianxin Wang, and Yi Pan. 2018. BridgeTaint: a bi-directional dynamic taint tracking method for JavaScript bridges in Android hybrid applications. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 677–692. https://doi.org/10.1109/TIFS.2018.2855650

[6] Emery D Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 4 (2019), 1–24. https://doi.org/10.1145/3340571

[7] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent program slicing. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120. https://doi.org/10.1145/2635868.2635893

[8] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Annual Computer Software and Applications Conference*. 303–312. https://doi.org/10.1109/COMPSAC.2013.55

[9] Robert Bradshaw, Stefan Behnel, Dag Sverre Seljebotn, et al. 2023. Cython: The Cython compiler for writing C extensions for the Python language. https://github.com/cython/cython GitHub repository.

[10] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. 2023. Generating vulnerable code via learning-based program transformations. In *AI Embedded Assurance for Cyber Systems*. Springer, 123–138. https://doi.org/10.1007/978-3-031-42637-7_7

[11] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu, and Kefei Wang. 2005. An empirical study of programming language trends. *IEEE Software* 22, 3 (2005), 72–79. https://doi.org/10.1109/MS.2005.55

[12] KR1442 Chowdhary and KR Chowdhary. 2020. Natural language processing. *Fundamentals of Artificial Intelligence* (2020), 603–649. https://doi.org/10.1007/978-81-322-3972-7_19

[13] Daniel P Delorey, Charles D Knutson, and Christophe Giraud-Carrier. 2007. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Second International Workshop on Public Data about Software Development (WoPDaSD'07)*. 1–5.

[14] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: fuzzing the binding code of JavaScript engines using semantically correct test cases. In *NDSS*. https://doi.org/10.14722/ndss.2021.24224

[15] Alessandro Fin, Franco Fummi, and Graziano Pravadelli. 2001. Amleto: A multi-language environment for functional test generation. In *Proceedings International Test Conference*. 821–829. https://doi.org/10.1109/TEST.2001.966704

[16] Python Software Foundation and contributors. 2023. cpython: The Python programming language. https://github.com/python/cpython/ GitHub repository.

[17] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 927–942. https://doi.org/10.1145/3385412.3386014

[18] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *International Conference on Mining Software Repositories*. 608–620. https://doi.org/10.1145/3524842.3528452

[19] Kovid Goyal. 2023. kitty: A cross-platform, fast, feature-rich, GPU based terminal emulator. https://github.com/kovidgoyal/kitty GitHub repository.

[20] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E Eghan, and Bram Adams. 2020. On the impact of interlanguage dependencies in multilanguage systems empirical case study on Java native interface applications (JNI). *IEEE Transactions on Reliability* 70, 1 (2020), 428–440. https://doi.org/10.1109/TR.2020.3024873

[21] Manel Grichi, Ellis E Eghan, and Bram Adams. 2020. On the impact of multi-language development in machine learning frameworks. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 546–556. https://doi.org/10.1109/ICSME46990.2020.00058

[22] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An extensible, regular-expression-based tool for multi-language mutant generation. In *IEEE/ACM International Conference on Software Engineering: Companion (ICSE-Companion)*. 25–28. https://doi.org/10.1145/3183440.3183485

[23] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[24] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. JustGen: effective test generation for unspecified JNI behaviors on JVMs. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 1708–1718. https://doi.org/10.1109/ICSE43902.2021.00151

[25] Capers Jones. 2010. *Software engineering best practices: lessons from successful projects in the top companies.* McGraw-Hill Education.

[26] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. 2016. A large scale study of multiple programming languages and code quality. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 563–573. https://doi.org/10.1109/SANER.2016.112

[27] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 649–660. https://doi.org/10.1109/ICSE43902.2021.00066

[28] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35 (2022), 21314–21328. https://doi.org/10.48550/arXiv.2207.01780

[29] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for JNI program analysis. In *IEEE/ACM International Conference on Automated Software Engineering*. 127–137. https://doi.org/10.1145/3324884.3416558

[30] Wen Li, Ming Jiang, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A cross-language dynamic information flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.

[31] Wen Li, Li Li, and Haipeng Cai. 2022. On the vulnerability proneness of multilingual code. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 847–859. https://doi.org/10.1145/3540250.3549173

[32] Wen Li, Li Li, and Haipeng Cai. 2022. PolyFax: A toolkit for characterizing multi-language software. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1662–1666. https://doi.org/10.1145/3540250.3558925

[33] Wen Li, Austin Marino, Haoran Yang, Na Meng, Li Li, and Haipeng Cai. 2024. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–46. https://doi.org/10.1145/3631967

[34] Wen Li, Na Meng, Li Li, and Haipeng Cai. 2021. Understanding language selection in multi-language software projects on GitHub. In *IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 256–257. https://doi.org/10.1109/ICSE-Companion52605.2021.00119

[35] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic greybox fuzzing of multi-language systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1379–1396.

[36] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *ACM SIGSOFT international symposium on software testing and analysis*. 169–180. https://doi.org/10.1145/3339068

[37] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault localization with code coverage representation learning. In *International Conference on Software Engineering (ICSE)*. 661–673. https://doi.org/10.1109/ICSE43902.2021.00067

[38] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Fault localization to detect co-change fixing locations. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 659–671. https://doi.org/10.1145/3540250.3549137

[39] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30. https://doi.org/10.1145/3360588

[40] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. 2021. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems* (2021), 6999–7019. https://doi.org/10.1109/TNNLS.2021.3084827

[41] Linus Eriksson. 2022. Tree-Sitter. https://github.com/tree-sitter/tree-sitter.

[42] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. 2020. FANS: Fuzzing Android native system services via automated interface analysis. In *USENIX Security Symposium*. 307–323.

[43] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. On the reproducibility and replicability of deep learning in software engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–46. https://doi.org/10.1145/3477535

[44] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676. https://doi.org/10.1145/3468264.3468580

[45] Philip Mayer and Alexander Bauer. 2015. An empirical analysis of the utilization of multiple programming languages in open source projects. In *International Conference on Evaluation and Assessment in Software Engineering*. 1–10. https://doi.org/10.1145/2745802.2745805

[46] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1. https://doi.org/10.1186/s40411-017-0035-z

[47] Larry R Medsker and LC Jain. 2001. Recurrent neural networks. *Design and Applications* 5 (2001), 64–67. https://doi.org/10.1201/9781003040620

[48] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *International Conference on Software Engineering*. 1169–1180. https://doi.org/10.1145/3510003.3510147

[49] Leo A Meyerovich and Ariel S Rabkin. 2013. Empirical analysis of programming language adoption. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 1–18. https://doi.org/10.1145/2509136.2509515

[50] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-code: sequence-to-sequence pre-training for learning source code representations. In *International Conference on Software Engineering*. 2006–2018. https://doi.org/10.1145/3510003.3510096

[51] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-scale sample generation for boosting learning-based software vulnerability analyses. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. Article 149, 13 pages. https://doi.org/10.1145/3597503.3639116

[52] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1097–1109. https://doi.org/10.1145/3540250.3549128

[53] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VulGen: Realistic vulnerability generation via pattern mining and deep learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2527–2539. https://doi.org/10.1109/ICSE48619.2023.00211

[54] OpenAI. 2019. ChatGPT: A Large-Scale Generative Model. https://openai.com/research/chatgpt. Accessed: 9/12/2023.

[55] Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. 2023. Static analysis of JNI programs via binary decompilation. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3089–3105. https://doi.org/10.1109/TSE.2023.3241639

[56] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *IEEE Symposium on Security and Privacy (SP)*. 2339–2356. https://doi.org/10.1109/SP46215.2023.10179420

[57] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating tree path in transformer for code representation. *Advances in Neural Information Processing Systems* 34 (2021), 9343–9354.

[58] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021), 1–22. https://doi.org/10.48550/arXiv.2105.12655

[59] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in GitHub. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165. https://doi.org/10.1145/3126905

[60] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155* (2018). https://doi.org/10.48550/arXiv.1803.02155

[61] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991. https://doi.org/aaai.v34i05.6430

[62] Federico Tomassetti and Marco Torchiano. 2014. An empirical assessment of polyglot-ism in GitHub. In *International Conference on Evaluation and Assessment in Software Engineering*. 1–4. https://doi.org/10.1145/2601248.2601269

[63] Sergi Valverde and Ricard V Solé. 2015. Punctuated equilibrium in the large-scale evolution of programming languages. *Journal of The Royal Society Interface* 12, 107 (2015), 20150249. https://doi.org/10.1098/rsif.2015.0249

[64] Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. 2013. The Babel of software development: Linguistic diversity in Open Source. In *International Conference on Social Informatics*. 391–404. https://doi.org/10.1007/978-3-319-03260-3_34

[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017), 1–15. https://doi.org/10.48550/arXiv.1706.03762

[66] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[67] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. RAP-Gen: retrieval-augmented patch generation with CodeT5 for automatic program repair. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158. https://doi.org/10.1145/3611643.3616256

[68] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Conference on Empirical Methods in Natural Language Processing*. 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[69] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–58. https://doi.org/10.1145/3485275

[70] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of Android applications with native code. In *ACM SIGSAC Conference on Computer and Communications Security*. 1137–1150. https://doi.org/10.1145/3243734.3243835

[71] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: state-of-the-art natural language processing. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[72] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. 2018. NDroid: Toward tracking information flows across multiple Android contexts. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 814–828. https://doi.org/10.1109/TIFS.2018.2866347

[73] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[74] Haoran Yang, Wen Li, and Haipeng Cai. 2022. Language-agnostic dynamic analysis of multilingual code: promises, pitfalls, and prospects. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1621–1626. https://doi.org/10.1145/3540250.3560880

[75] Haoran Yang, Weile Lian, Shaowei Wang, and Haipeng Cai. 2023. Demystifying issues, challenges, and solutions for multilingual software development. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 1840–1852. https://doi.org/10.1109/ICSE48619.2023.00157

[76] Haoran Yang, Yu Nong, Shaowei Wang, and Haipeng Cai. 2024. Multi-language software development: issues, challenges, and solutions. *IEEE Transactions on Software Engineering* 50, 3 (2024), 512–533. https://doi.org/10.1109/TSE.2024.3358258

[77] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. 2023. Declarative static analysis for multilingual programs using CodeQL. *Software: Practice and Experience* 53, 7 (2023), 1472–1495. https://doi.org/10.1002/spe.3199

[78] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Xiaohong Zhang. 2021. A study of effectiveness of deep learning in locating real faults. *Information and Software Technology* 131 (2021), 106486. https://doi.org/10.1016/j.infsof.2020.106486

[79] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

Received 2023-09-28; accepted 2024-04-16