CrossProbe: LLM-Empowered Cross-Project Bug Detection for Deep Learning Frameworks

HAO GUAN[*], University of Queensland, Australia and Southern University of Science and Technology, China
GUANGDONG BAI[†], University of Queensland, Australia
YEPANG LIU[†‡], Southern University of Science and Technology, China

Deep Learning (DL) models may introduce reliability challenges in the underlying DL frameworks. These frameworks may be prone to bugs that can lead to crash or wrong results, particularly when involving complex model architectures and substantial computational demands. Such framework bugs can disrupt DL applications, impacting customer experience and potentially causing financial losses. Traditional approaches to testing DL frameworks face limitations in adapting to the vast search space of model structures, diverse APIs, and the complexity of hybrid programming and hardware environments. Recent advancements using Large Language Models (LLMs) have improved DL framework fuzzing, but their efficacy depends heavily on the quality and diversity of input prompts, which are often constructed using single-framework data.

In this paper, we propose an innovative approach for enhancing test generation for DL frameworks by leveraging "mirroring issues"—analogous bugs identified across different frameworks with common functionalities. Our approach is inspired by the fact that DL frameworks, such as PyTorch and TensorFlow, often share common bugs due to dependencies, developer errors, or edge-case inputs. We develop CrossProbe that utilizes LLMs to effectively learn from existing issues of one framework and transfer the acquired knowledge to generate test cases for finding mirroring issues in another framework, thus enabling cross-framework bug detection. To overcome the challenges of test case generation arising from the incompatible functionalities and different implementations between frameworks, we introduce three processes: *alignment*, *screening*, and *distinction*. These processes help mitigate transfer errors by establishing API pair databases, filtering unsuitable cases, and highlighting cross-framework distinctions. Experiments demonstrate that CrossProbe is efficient by saving 36.3% iterations of generation, and achieves a 25.0% higher success rate in issue transferring compared to existing state-of-the-art LLM-based testing techniques. CrossProbe detects 24 unique bugs using its transferred knowledge. Out of them, 19 are previously unknown and each requires cross-framework knowledge in deep learning for identification.

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

Additional Key Words and Phrases: Deep Learning Frameworks, Library Testing, Large Language Model

[*]Hao Guan is under the UQ-SUSTech Joint PhD Program.
[†]The corresponding authors are Yepang Liu and Guangdong Bai.
[‡]Yepang Liu is affiliated with the Research Institute of Trustworthy Autonomous Systems and Department of Computer Science and Engineering at Southern University of Science and Technology.

Authors' Contact Information: Hao Guan, University of Queensland, Brisbane, Australia and Southern University of Science and Technology, Shenzhen, China, hao.guan@uq.edu.au; Guangdong Bai, University of Queensland, Brisbane, Australia, g.bai@uq.edu.au; Yepang Liu, Southern University of Science and Technology, Shenzhen, China, liuyp1@sustech.edu.cn.

ACM 2994-970X/2025/7-ARTISSTA107
https://doi.org/10.1145/3728984

ACM Reference Format:
Hao Guan, Guangdong Bai, and Yepang Liu. 2025. CrossProbe: LLM-Empowered Cross-Project Bug Detection for Deep Learning Frameworks. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA107 (July 2025), 23 pages. https://doi.org/10.1145/3728984

1 Introduction

Deep Learning (DL) techniques have become a widespread solution to tasks like classification and generation due to their impressive abilities. In the last decade, DL models are widely used across multiple fields and domains, such as medical diagnosis [2, 30], autonomous driving [20], and natural language processing [38]. These models often rely on a large number of parameters and deep architectures to handle complex problems and achieve high accuracy. As a result, the operation of DL models, including training, inference, and optimization, can be computationally intensive and vulnerable to errors. In particular, the bugs inside frameworks can severely compromise the reliability of deep learning applications or even hinder their deployment on users' devices, resulting in negative impacts on customer experience, financial losses and, in some cases, even physical injury or fatalities [5, 12, 17].

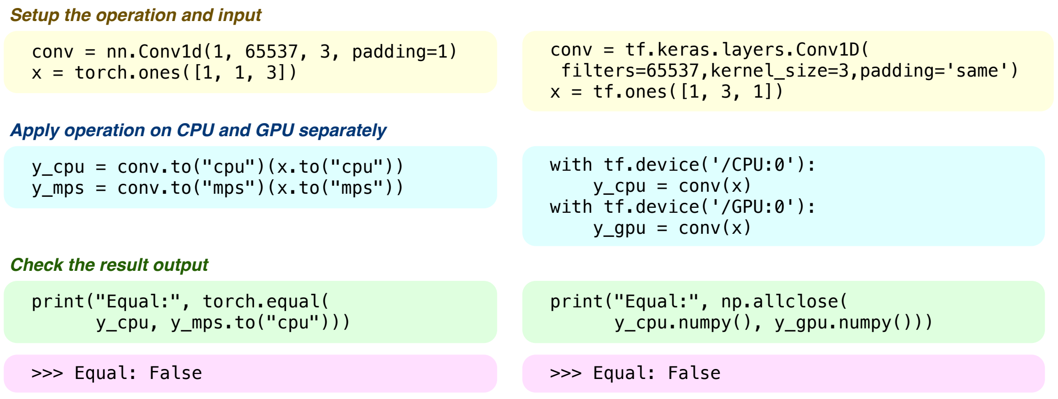
To test deep learning frameworks, the testing cases usually involve an input model and tensor operations on the model. Traditional approaches show limitations when generating test cases for deep learning frameworks. The integration of hybrid programming languages and the need to support diverse hardware and platforms pose substantial difficulties for approaches relying on static analysis [25, 26]. Besides, it is also complex to dynamically generate test cases due to the huge search space of model structures and operation APIs [7, 14]. Due to such challenges, existing approaches either restrict the searching within specific DL models [15, 16, 41, 48, 49] or focus on a partial subset of APIs [9, 11, 21].

Recently, Large Language Models (LLMs) have demonstrated remarkable capabilities in enhancing the testing of deep learning frameworks [7, 8, 13, 45], by generating precise and targeted test cases. The core of LLM-based generation lies in the quality of input prompts. Existing approaches often refer to various sources of data of the framework under test to mine key information for prompt construction, including historical issues [8, 13], open-source examples [21] and documentation [45, 46]. The mined information is then formatted using prompting techniques such as chain-of-thought [43], few-shot learning [40], and retrieval-augmented generation [18] to help LLMs learn the characteristics from existing data and generate new test cases. However, these approaches may encounter bottlenecks when the available data is insufficient. As test iterations progress, relying on information from a single framework may no longer be adequate to uncover untested parts inside the framework. Additionally, the knowledge learned by LLMs is specific to a single framework and cannot be effectively transferred across frameworks. These challenges highlight the need for leveraging diverse and transferable information to enhance LLM-based test generation, ultimately enabling comprehensive testing of deep learning frameworks.

Mirroring issues. We extend the scope of information mining from a framework itself to its analogous frameworks, inspired by a key insight that frameworks providing analogous functionalities often share analogous types of bugs. Take a numerical bug shown in Figure 1 as an example. In PyTorch (left), discrepancies have been reported between computations executed on a CPU and an Apple GPU (MPS). When the same computation is ported to TensorFlow (right), analogous discrepancies also arise. We refer to such pairs of issues as *mirroring issues*, which are characterized by 1) their involvement with analogous input data and API functionalities, and 2) their ability to trigger analogous bugs in two different frameworks. Mirroring issues prevalently exist in frameworks with analogous purposes due to several factors:

- *similar dependencies* that different frameworks rely on,

- *common mistakes* that developers may make, due to their analogous interpretations of the problem domain, and
- *shared edge cases* that occur in analogous functionality and tend to be overlooked during testing.


Setup the operation and input
Apply operation on CPU and GPU separately
Check the result output

```
conv = nn.Conv1d(1, 65537, 3, padding=1)
x = torch.ones([1, 1, 3])
```

```
conv = tf.keras.layers.Conv1D(
 filters=65537,kernel_size=3,padding='same')
x = tf.ones([1, 3, 1])
```

```
y_cpu = conv.to("cpu")(x.to("cpu"))
y_mps = conv.to("mps")(x.to("mps"))
```

```
with tf.device('/CPU:0'):
    y_cpu = conv(x)
with tf.device('/GPU:0'):
    y_gpu = conv(x)
```

```
print("Equal:", torch.equal(
      y_cpu, y_mps.to("cpu")))
```

```
print("Equal:", np.allclose(
      y_cpu.numpy(), y_gpu.numpy()))
```

```
>>> Equal: False
```

```
>>> Equal: False
```

Fig. 1. An Example of Mirroring Issues between PyTorch and TensorFlow

Our work. In this work, we generate mirroring issues from existing issues in one DL framework to enhance test case generation for another DL framework. Our approach starts with identifying issues that involve analogous functionalities in both frameworks as the candidates for generating mirroring issues. We then leverage LLMs to transfer these candidate issues into code that provides the analogous functionality but is compatible with the other framework. This transfer process faces significant challenges that can result in low success rate for valid code generation, due to differences in code implementation across frameworks. The transferred code may suffer from API misuse, incorrect logic, or missing necessary statements, leading to false positives and false negatives in the testing process. Specifically, we observe during code transfer, LLMs frequently encounter the following two major obstacles:

Obstacle #1. Significant adaption gap for functionalities. The first primary obstacle in transferring code between frameworks is the difficulty in selecting the appropriate APIs. Although different deep learning frameworks share some core functionalities, it requires specialized knowledge to figure out the corresponding APIs for different frameworks, because the naming conventions and design patterns usually vary. Besides, a number of exclusive APIs or data types remain incompatible across frameworks, further complicating the transfer process. For instance, PyTorch and TensorFlow, the two most widely-used DL frameworks, offer the same core functionalities such as tensor operations and creating a model from layers. However, their differences in distributed structures and data processing workflows are substantial. The involvement of different packages, such as torchvision and Keras, renders the porting process complex. It can go beyond the capability of LLMs to adapt to the significant differences. As a result, the test cases generated by LLMs can be invalid, failing to reflect the original behavior of the source issues.

Obstacle #2. Subtle usage discrepancies in APIs. Even when the appropriate API is identified, the transfer remains challenging due to the subtle differences in API usage. For example, as illustrated in Figure 1, PyTorch and TensorFlow adopt distinct coding styles for GPU acceleration. These differences are not merely syntactic but extend to variations in parameter order, naming conventions, default values, and input data formats. Without a precise understanding of API definitions and their intended functionalities, the generated code may be prone to critical errors, which can degrade the efficiency of the testing. To enhance the accuracy of code generation, LLMs should be enhanced with contextual awareness of API usage and the capability of adapting to varied coding conventions.

To mitigate these obstacles and enhance the quality of generated test cases, we introduce three key processes: *alignment*, *screening* and *distinction*. They leverage knowledge of functionality-equivalent APIs across frameworks to facilitate accurate code transfer:

Alignment is the data mining and analysis process for the target pair of frameworks. Since API discrepancies are the primary challenge in code transfer, identifying the most appropriate API replacements is crucial. To this end, we construct a database of API pairs with analogous functionalities and usages across framework. We train a classification model from the framework documentation based on CodeT5 [39] to characterize the APIs more accurately, and produce a more comprehensive database compared with the simple keyword matching approach. This database serves as an effective reference for adapting the code differences.

Screening is designed to filter inappropriate issues to reduce the risk associated with ***Obstacle #1***. Using the database constructed by *alignment*, we can estimate the functionality gap to adapt the code from one framework to the other. For a candidate issue, if the corresponding API is not found in the alignment database, the adaption gap is considered huge and the likelihood for LLMs to generate a valid test case decreases significantly. Therefore, we discard the input issues with significant adaption gap, and the cases with APIs in the alignment database are preferred.

Distinction provides prompts enriched with structured knowledge that guides LLMs in adapting code in a chain-of-thought manner. These prompts highlight the similarities and differences between API pairs. They reference the names of aligned API pairs for LLMs to make appropriate API substitutions, and include comparisons and examples of API usage that enable LLMs to adapt the code by converting each parameter and following sample code step by step. These prompts can assist LLMs in addressing ***Obstacle #2*** and improving the accuracy of transferred code.

The above processes are implemented as CrossProbe. To evaluation its effectiveness, we compare it with the state-of-the-art LLM-based testing techniques, including FuzzGPT [8] and YanHui [13]. CrossProbe achieves a 25.0% increase in the success rate of transferring issue code. We also investigate its generation efficiency by conducting an ablation study. The study shows that CrossProbe can detect three times as many bugs with 36.3% fewer generated test cases, compared with the baseline approach without our designed processes. We review and characterize the bugs detected by CrossProbe, and find 19 previously-unknown bugs. They are are difficult to trigger with the knowledge of one single framework, and CrossProbe manages to expose them with the knowledge transferred from other frameworks.

Contributions. This work makes the following main contributions:

- **The introduction of cross-framework knowledge to the testing of DL frameworks**. Our study reveals that, with proper adaptation, test cases from one DL framework can be effectively used to test other DL frameworks, resembling practices in compiler and network protocol testing [24, 45]. We characterize the types of bugs that can be exposed by cross-framework testing, particularly those related to shared dependencies and complex operations.
- **An effective LLM-based approach for cross-framework bug detection**. We develop CrossProbe, a practical framework to learn from the issues in one framework, and transfers the tests to detect bugs in another framework. CrossProbe overcomes the obstacles in API selection and usage adaptation during code transfer, and produces a high rate of correct code to achieve effective cross-framework bug detection.
- **Complementary defect discovery compared with fuzzing**. CrossProbe has detected 19 bugs, which are adapted from the cases in other frameworks. Fuzzing approaches are difficult to cover such cases by mutation, because the knowledge of potential testing points is often

limited to the target framework itself. Cross-framework test cases can enhance the framework testing by introducing testing points from other frameworks. As of our submission, **ten bugs have been confirmed by developers, three are resolved**, and six are awaiting responses.

2 Preliminaries

2.1 Code Generation With Large Language Models

Large Language Models (LLMs) [10, 27, 29, 35, 37, 38] are deep learning models that can accomplish natural language processing, including text classification, summary and generation. As the amount of training text increases rapidly, the models keep evolving and have demonstrated impressive capabilities on the NLP tasks. The recent popular LLM model series, Llama [1], contains up to 405 billion of parameters in version 3.2. To accomplish code-specific tasks, some variants of LLMs are introduced by adding code into training data, or fine-tuning the original model with code-related contexts. Typical code-specific LLMs include CodeT5 [39], CodeLlama [32], and StarCoder [22].

LLMs can accept inputs in the form of prompts [31, 44], which is a flexible and light-weight manner because the expensive retraining process of models can be avoided. To improve the quality of the generated code, various prompting techniques have been proposed to arouse the reasoning capability of LLMs. This technique involves dividing each prompt into intermediate segments that explicitly incorporate the reasoning process, allowing for greater clarity and structure in task execution. By leveraging these incremental steps, this approach has been demonstrated to significantly enhance the accuracy of the responses generated by LLMs [8, 13, 19, 47]. For instance, intermediate segments may include specific instructions on processing previous results, detailing the necessary formats, or outlining the constraints that must be adhered to when generating subsequent content. These well-defined guidelines ensure that the model has a comprehensive understanding of each stage of the task, reducing ambiguity and minimizing the potential for errors. By adhering to this structured approach, LLMs are better equipped to generate responses that are consistent with the desired outcomes. Consequently, this structured prompting methodology ensures that the final outputs are more likely to meet the users' expectations, providing an effective pattern for handling professional requirements.

Another effective prompting technique is to utilize few-shot examples, which may offer some specific or implicit requirements for code generation. By providing a set of relevant examples, this method guides the LLMs in understanding the desired patterns and standards, helping them produce more accurate and contextually appropriate outputs. For instance, FuzzGPT [8] provides the information of API, bug description, and finally code snippets as the steps of thought. YanHui [13] provides more comprehensive insights into the characteristics of buggy code snippets, covering aspects such as model structure, exceptions, data types, and error causes.

Existing methods have achieved outstanding performance in code generation within a single project. However, the generation may fall short when it comes to transferring code between different projects. We illustrate the possible challenges through the following motivational study.

2.2 Motivational Study

To characterize mirroring issues in deep learning frameworks and understand the challenges in detecting them, we conduct an empirical study on the two mainstream deep learning frameworks, PyTorch and TensorFlow.

2.2.1 Characterizing Mirroring Issues in PyTorch and TensorFlow.

Dataset construction. We first collect issues from PyTorch and TensorFlow automatically via the GitHub API. We choose to search mirroring issues since the following latest major releases of PyTorch and TensorFlow: PyTorch 2.4 (released on July 25, 2024) and TensorFlow 2.17.0 (released on July 12, 2024). We restrict our issue search scope for two primary reasons. First, we aim to study recent issues that may have a larger impact compared to issues in older versions. Second, the latest major releases provide stable and comprehensive features of PyTorch and TensorFlow. There are already sufficient common functionalities between the two frameworks and a large number of existing issues for our study. The cutoff date for our data collection is December 31, 2024. To construct the dataset, we extract code from the following sources and only select the issues for which we can find reproduction code.

- Code blocks in the issue reports
- External links to code snippets
  - Gist: A popular GitHub service to share simple code snippets.
  - CoLab: A popular platform to run Jupyter Notebook online.

Initially, we collect a total of 1,452 issues, with 813 from PyTorch and 639 from TensorFlow. Then, we conduct an iterative labeling process following the open coding methodology [36] to produce a comprehensive characterization of mirroring issues.

Iteration 1: We study each of the collected issues and summarize the core APIs and the concerned functionality based on the code in the issues. This is a semi-automatic process, where we first extract the API names and retrieve the corresponding documentation automatically. Then, we manually summarize the main functionality of the bug reproduction code. With the summarization, we match the pairs of PyTorch and TensorFlow issues with analogous functionalities and core APIs, which are identified as mirroring issues between the two frameworks.

Iteration 2: Another author is involved to cross-validate the preliminary result obtained in *Iteration 1* and check whether the functionality description confirms to the bug reproduction code for each issue. For the issues where this author does not agree with the preliminary result, the two authors discuss them and clarify the criteria used in the analysis process. With the updated understanding of the analysis criteria, the second author applies another round of validation to check whether a new disagreement is introduced. Several rounds of discussions have been conducted to reduce the disagreements.

Iteration 3: After the above calibration of the analysis process, all authors revisit each bug and discuss to reach agreement on every pair of mirroring issues to make sure that the identified pairs indeed reflect analogous problems in the two frameworks.

Table 1. Mirroring Issues Collected from PyTorch and TensorFlow

Total Pairs	Environment	Configuration	Computation	Format
74	25	22	19	8

Characterization results. Table 1 presents the distribution of the mirroring issues collected. In total, we have identified 74 pairs of mirroring issues from the latest major releases of PyTorch and TensorFlow, which consist of the following four categories:

- **Environment:** Issues arising from the runtime environment, including installed packages, operating systems, and GPU hardware. For example, TensorFlow #85689 and PyTorch #143123 are mirroring issues related to the environment.
- **Configuration:** Issues triggered by parameter settings, custom operator behaviors, and data type or precision configurations. For example, <PyTorch #101620, TensorFlow #65865> and <PyTorch #108520, TensorFlow #65871> are two pairs of mirroring issues of this category.
- **Computation:** As operations like tensor computations are common to both frameworks, mirroring issues can arise from certain common computations. For example, PyTorch #117757 and TensorFlow #62517 are mirroring issues related to the `abs` operator.

- **Formats:** Issues that occur during the I/O process or the model format conversion. For example, PyTorch #143222 and TensorFlow #77293 are such mirroring issues.

2.2.2 *Challenges in Detecting Mirroring Issues via Transferring Issue-Triggering Code.*

Basic CoT approach. To reveal the challenges of detecting mirroring issues between two frameworks via transferring issue-triggering code (or reproduction code) using LLMs, we design a basic approach, which involves simple chain-of-thought [43] prompts to convert the code that triggers issues in one framework to the other framework. This approach is referred to as Basic CoT in the subsequent discussions.

To understand the performance of Basic CoT, we randomly select 50 pairs of mirroring issues from our dataset. Then we provide prompts, which follow the format demonstrated in Listing 1, to LLMs to convert the issue-triggering code from PyTorch to TensorFlow and vice versa.

```
# Here is a code snippet which can trigger a bug in PyTorch
(code...)
# Please convert the code into TensorFlow to find potential bugs.
<infill>
```

Listing 1. Prompt Template for the Basic CoT Approach

To assess the quality of the 100 code generation results from the Basic CoT approach, we consider the following two criteria.

1) Syntactically valid outputs. The generated code is valid Python code with no syntax errors. We filter out invalid outputs by applying the PyLint static analyzer first, and then actually running the generated code and excluding those cases that produce exceptions of `NameError`, `SyntaxError` and `ModuleNotFoundError`. This criterion can help assess whether a generation approach has the ability to produce valid code for the target framework.

2) Functionality-preserving outputs. Besides being syntactically correct, the code generated for the target framework also follows the original functionalities of the issue-triggering code for the source framework, allowing for the detection of potential bugs in the target framework. This criterion is designed to measure the approach's ability of recognizing the API from the test cases for the source framework and using the corresponding API in the target framework to generate functionally equivalent test cases.

Table 2. Code Generation Quality of The Basic CoT Approach

Model	**Outputs**	**Syntactically Valid**	**Functionality Preserved**
gpt-4-0806	100	18	12
codellama-py-7b	100	14	11

Results. Table 2 demonstrates the results of Basic CoT when using two popular LLMs. It can be seen that Basic CoT exhibits a low success rate and limited effectiveness in transferring code across frameworks. We manually analyze the failed transfers by sampling 53 of the 168 invalid code outputs. Our analysis shows that with only simple chain-of-thought prompts, LLMs would make the following mistakes when generating test cases for the target framework: **using nonexistent APIs** (43.4%), **using wrong arguments of APIs** (30.2%), and **incomplete code** (26.4%). Besides, among the nine transfer results with valid syntax but wrong functionalities, six are **not following the original behaviors** (data or operations) and three are **incomplete code** leaving some functionality unimplemented. From the sampled cases, we observe that the poor performance of Basic CoT can be attributed to two major reasons:

First, successful code transfer involves accurately mapping or aligning APIs between the frameworks, which is a prerequisite for reliably generating functionally-equivalent test cases across frameworks (***Obstacle #1** mentioned in introduction*). This obstacle may cause nonexistent APIs, wrong arguments and incomplete code. For example, PyTorch has the `torch.distributed.ProcessGroup` API to communicate among devices, while TensorFlow implements a different design — the training coordinator and worker strategy. When converting an issue in PyTorch, LLMs fail to follow TensorFlow's distributed training strategy and use correct APIs. Instead, the generated code snippets try to imitate the structure of the code that triggers the issue in PyTorch and create a non-existent custom class called `MockProcessGroup`, which makes no sense to the user.

Second, in practice, it is common that different frameworks may impose unique conventions, API parameters, or structural designs (e.g., inheritance vs. composition). It is non-trivial to generate code adhering to the correct usage patterns required by the concerned APIs in the target framework (***Obstacle #2***). This obstacle may cause wrong arguments, incomplete code, or the transferred code not following the original behaviors. For example, PyTorch and TensorFlow are using different ways to specify the computation device. While TensorFlow simply uses a string such as `"GPU:1"`, PyTorch requires configurations on every Tensor class for the same purpose. The LLM-generated code for PyTorch may partially miss the `.to()` API call and corresponding arguments.

Motivated by the above observations, we aim to design a more effective approach for transferring test cases between different frameworks to detect mirroring issues.

3 Methodology

To overcome the obstacles of effective code transfer across frameworks, we introduce three key processes that are designed to distill and integrate transferrable knowledge to guide LLMs in producing accurate outputs. The processes are termed as ***alignment***, ***screening***, and ***distinction***. This section begins with a general overview of our approach, CrossProbe, followed by a detailed explanation of each process.

3.1 Overview

Figure 2 demonstrates the overall workflow of CrossProbe. Given two frameworks with analogous functionalities (e.g., PyTorch and TensorFlow), CrossProbe first performs API alignment via referring to the framework documentation and establish a precise mapping between the functionalities of the two frameworks. The mapping serves as the knowledge database to facilitate code transfer by providing *which APIs from the both frameworks are suitable for transfer*, and *how to use the APIs correctly*. Second, CrossProbe takes the issues from one framework (source) as input and applies screening to identify the useful issue-triggering code for transfer, where *the APIs are included in the alignment database*. Then, distinction provides effective prompts to generate the test cases for the other framework (target). The prompt contains *the description and usage of the APIs from both frameworks* and leverages chain-of-thought technique to help LLMs to generate correct transferred code. Finally, The generated code is executed to test the target framework.

3.2 API Alignment

In the alignment process, CrossProbe aims to identify the APIs with analogous functionalities from the two frameworks and extract their usages to establish an API mapping (i.e., pairs of analogous APIs). For this purpose, CrossProbe first processes the documentation of each framework and retrieves the API names and descriptions for analysis. When processing the documentation of PyTorch and TensorFlow, we find that the two frameworks have some obviously analogous APIs, which are named after their common operations. Some examples are listed below.

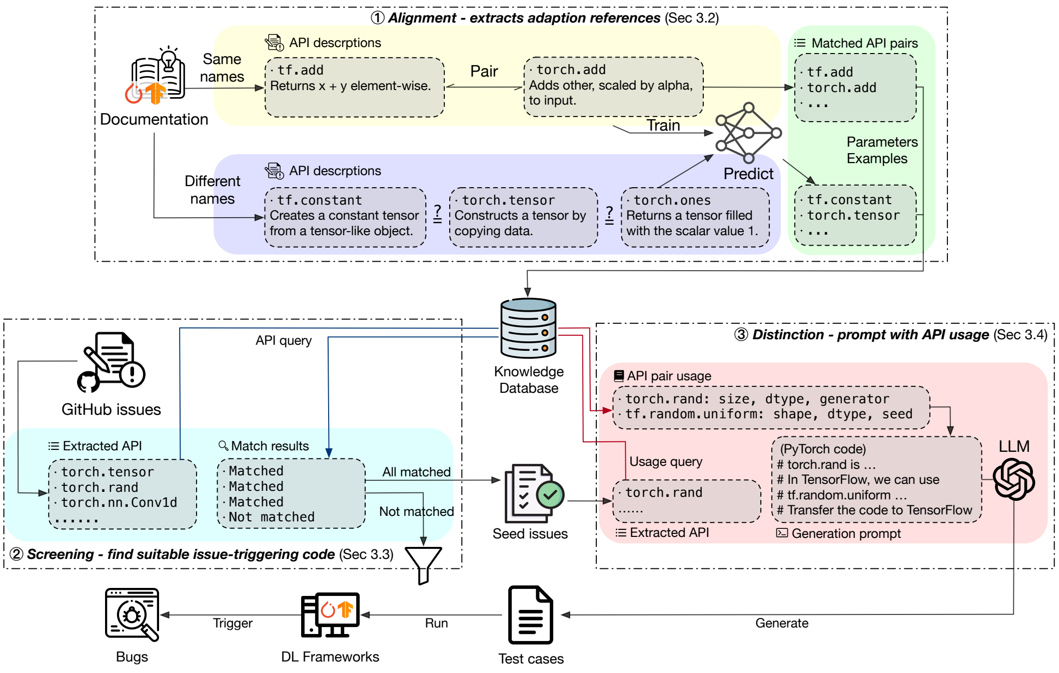

① Alignment - extracts adaption references (Sec 3.2)
Documentation
Same names
Different names
API descrptions
tf.add Returns x + y element-wise.
Pair
torch.add Adds other, scaled by alpha, to input.
Train
Predict
tf.constant Creates a constant tensor from a tensor-like object.
torch.tensor Constructs a tensor by copying data.
torch.ones Returns a tensor filled with the scalar value 1.
Matched API pairs
tf.add torch.add ...
Parameters Examples
tf.constant torch.tensor ...
Knowledge Database
GitHub issues
API query
Extracted API
torch.tensor torch.rand torch.nn.Conv1d ......
Match results
Matched Matched Matched Not matched
All matched
Not matched
Seed issues
② Screening - find suitable issue-triggering code (Sec 3.3)
③ Distinction - prompt with API usage (Sec 3.4)
API pair usage
torch.rand: size, dtype, generator
tf.random.uniform: shape, dtype, seed
Usage query
torch.rand ......
Extracted API
(PyTorch code) # torch.rand is ... # In TensorFlow, we can use # tf.random.uniform ... # Transfer the code to TensorFlow
Generation prompt
LLM
Bugs
Trigger
DL Frameworks
Run
Test cases
Generate

Fig. 2. Overall Workflow of CrossProbe

- Tensor values: `torch.ones` and `tf.ones`
- Tensor operations: `torch.add` and `tf.add`
- Neural networks: `torch.nn.ReLU` and `tf.nn.ReLU`
- Optimization algorithms: `torch.optim.SGD` and `tf.keras.optimizers.SGD`

After manual verification, we find that such APIs are indeed analogous and can be easily paired with each other by dropping the package prefix (`torch.` and `tf.`) of the API names and matching the remaining parts. However, there also exist analogous APIs with different names. For these cases, we measure the API similarity by calculating the cosine difference of the embedding of the APIs' functionality description. Specifically, we have trained a model with BERT to determine whether two given APIs are analogous by analyzing their descriptions. For model training, we leverage the descriptions of the APIs that are matched by API names.

With the trained model, we have found some APIs with analogous functionalities but different names according to the API description. Then we store the name, parameters and code examples of each API pair into the knowledge database, as illustrated in Figure 3. The example of such API pairs are presented as follows:

- Tensor construction: `torch.tensor` and `tf.constant`
- Random tensors: `torch.rand` and `tf.random.uniform`
- Loss functions: `torch.nn.MSELoss` and `tf.keras.losses.MeanSquaredError`

With the alignment process, we have collected 427 analogous API pairs from PyTorch and TensorFlow. Among them, 298 pairs are identified by matching API names, and 129 pairs are found with the similarity model. These API pairs have covered 28.9% of PyTorch APIs and 20.2% of TensorFlow APIs. For the API pairs, we will further extract the information on parameters and examples in the documentation and integrate the knowledge into the prompts in the distinction process (Section 3.4).

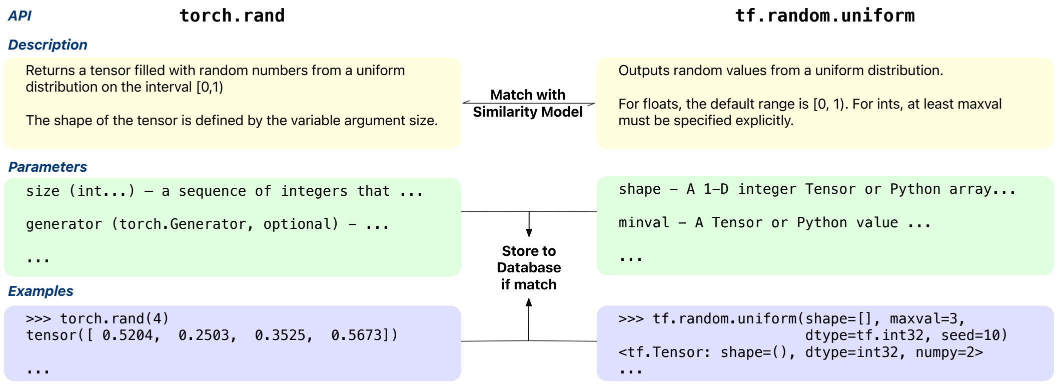

API
torch.rand
tf.random.uniform
Description
Returns a tensor filled with random numbers from a uniform distribution on the interval [0,1)
The shape of the tensor is defined by the variable argument size.
Match with Similarity Model
Outputs random values from a uniform distribution.
For floats, the default range is [0, 1). For ints, at least maxval must be specified explicitly.
Parameters
size (int...) – a sequence of integers that ...
generator (torch.Generator, optional) – ...
...
shape – A 1-D integer Tensor or Python array...
minval – A Tensor or Python value ...
...
Store to Database if match
Examples
>>> torch.rand(4)
tensor([0.5204, 0.2503, 0.3525, 0.5673])
...
>>> tf.random.uniform(shape=[], maxval=3, dtype=tf.int32, seed=10)
<tf.Tensor: shape=(), dtype=int32, numpy=2>
...

Fig. 3. An Example of Aligned API Pair

3.3 Issue Screening

The screening process is to filter out the inappropriate issue-triggering code from the issues in source framework and improve the success rate of transfer. We consider a piece of code inappropriate to be transferred if the APIs involved in the code do not have analogous ones in the target frameworks. As discussed in Section 1, transfer failures can be caused by converting APIs that have huge differences between frameworks, and the statistics in Section 2.2 shows that 58% of the failed conversions by Basic CoT originate from unmatched API calls. The abnormally low conversion success rate (below 15%) can introduce substantial noises that complicate subsequent bug validation processes. In addition, the unmatched APIs usually represent functional divergences between different frameworks. However, our work primarily focuses on mirroring issues that usually occur under targeted inputs for analogous functionalities. These considerations motivate us to implement rigorous filtering in CrossProbe to improve both computational efficiency and the quality of the transferred code, ultimately enhancing bug detection accuracy.

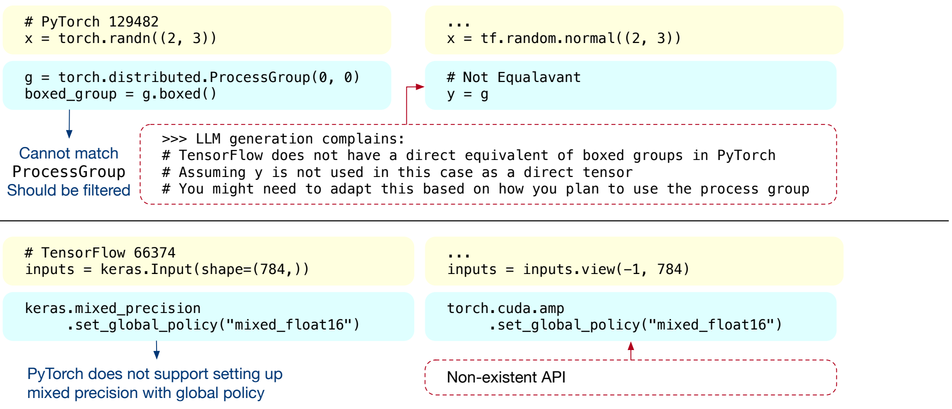
For issue screening, CrossProbe extracts the APIs used in the triggering code of the source issue, and query them in the database constructed in Section 3.2. The issue-triggering code with no matching APIs will be excluded from further conversion. For example, the code that involves the following exclusive APIs will be excluded.

- Platform specific: `torch.distributed.ProcessGroup`
- Special operation: `keras.mixed_precision.set_global_policy`

Such APIs will easily cause transfer failures due to the lack of corresponding functionality in the target framework, or lead to the generation of wrong code. For illustration, Figure 4 shows two examples with mismatched APIs. In the code transferred from PyTorch #129482, the API call `torch.distributed.ProcessGroup` will be converted to a blank implementation by LLM, with comments complaining the lack of directly equivalent API. TensorFlow #66374 presents another scenario where the transferred code is not even syntactically correct. The special operator `keras.mixed_precision.set_global_policy` will cause LLMs to generate code involving non-existent APIs in PyTorch.

3.4 API Usage Distinction

With the API database constructed in the alignment step and the seed issues identified via the screening process, we further design API usage distinction prompts to guide LLMs to accurately transfer issue-triggering code. The distinction prompts contain the usage information of the APIs from both source and target frameworks, and leverage the chain-of-thought technique to effectively



```
# PyTorch 129482
x = torch.randn((2, 3))

g = torch.distributed.ProcessGroup(0, 0)
boxed_group = g.boxed()
```

```
...
x = tf.random.normal((2, 3))

# Not Equalavant
y = g

>>> LLM generation complains:
# TensorFlow does not have a direct equivalent of boxed groups in PyTorch
# Assuming y is not used in this case as a direct tensor
# You might need to adapt this based on how you plan to use the process group
```

```
# TensorFlow 66374
inputs = keras.Input(shape=(784,))

keras.mixed_precision
     .set_global_policy("mixed_float16")
```

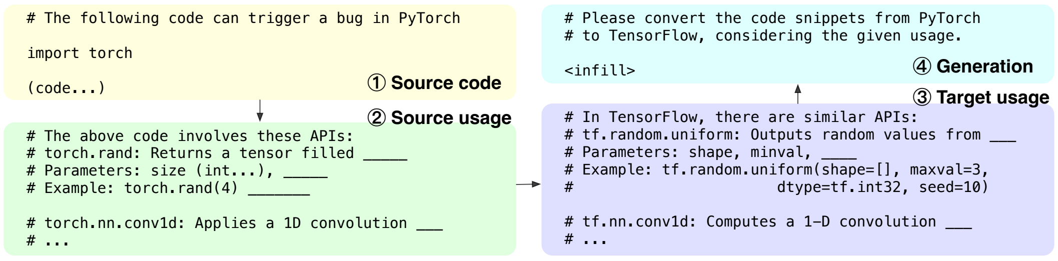
PyTorch does not support setting up
mixed precision with global policy

```
...
inputs = inputs.view(-1, 784)

torch.cuda.amp
     .set_global_policy("mixed_float16")
```

Fig. 4. Examples of Filtered Issues with Mismatched APIs

provide necessary knowledge to LLMs. Specifically, CrossProbe integrates the following two types of API usage information in prompts, which can be retrieved from the API database (Section 3.2). **1) API parameters**. The documentation explains the type, order or value for the parameters of the API. This kind of information will help LLMs to prepare the appropriate data to be used as API input, and avoid making parameter-related mistakes when generating API calls (e.g., wrong order). **2) Code examples**. The short code snippets provide more context on the use case of the API. In particular, the type and expected value of the API output are demonstrated, which can help LLMs to know about the functionality of the API.



```
# The following code can trigger a bug in PyTorch

import torch

(code...)
```

```
# The above code involves these APIs:
# torch.rand: Returns a tensor filled _____
# Parameters: size (int...), _____
# Example: torch.rand(4) _______

# torch.nn.conv1d: Applies a 1D convolution ___
# ...
```

```
# Please convert the code snippets from PyTorch
# to TensorFlow, considering the given usage.

<infill>
```

```
# In TensorFlow, there are similar APIs:
# tf.random.uniform: Outputs random values from ___
# Parameters: shape, minval, ____
# Example: tf.random.uniform(shape=[], maxval=3,
#                       dtype=tf.int32, seed=10)

# tf.nn.conv1d: Computes a 1-D convolution ___
# ...
```

Fig. 5. An Example of Prompt with Distinction Context

As illustrated in Figure 5, the distinction process employs chain-of-thought prompting to provide guidance on usage patterns for both frameworks. LLMs are prompted to systematically analyze and compare usage patterns, thereby encouraging them to account for differences between the APIs of different framework. Specifically, the PyTorch source code in Figure 5 contains APIs of `torch.rand` and `torch.nn.conv1d`. The distinction provides the usage information of these two APIs, as well as the corresponding TensorFlow APIs matched from the alignment database, including `tf.random.uniform` and `tf.nn.conv1d`. By learning from the structured comparison information, LLMs are better guided to produce code that aligns with the correct usage of the target framework. For example, the prompts in Figure 5 imply that the corresponding API of `torch.rand` is `tf.random.uniform` in TensorFlow, and the name of the first argument is "shape" rather than "size". Such specific knowledge helps mitigate the pitfalls during code transfer, particularly those related to API misalignment and misuses, and enhance the validity of the generated test cases.

4 Evaluation

In this section, we evaluate our approach by investigating the following three research questions.

- ***RQ1 (Effectiveness of CrossProbe): Can CrossProbe effectively transfer issue-triggering code across frameworks? How does it compare with the state-of-the-art LLM-based code generation approaches?*** This RQ aims to investigate the quality of the test cases generated by CrossProbe and perform comparisons with Basic CoT as well as recently proposed LLM-based deep learning library testing approaches [8, 13].
- ***RQ2 (Ablation Study): How do the three key steps of CrossProbe contribute to the improvement of code transfer quality?*** This RQ conducts an ablation study to understand how the alignment, screening, and distinction processes affect CrossProbe's test case generation.
- ***RQ3 (Usefulness): Is the transferred code by CrossProbe useful to find bugs inside deep learning libraries?*** This RQ focuses on analyzing whether the transferred code can help detect real bugs, and reveals the practical value of leveraging mirroring issues for library testing.

4.1 Experiment Design

Experiment Platform. The experiments are conducted on a workstation with the following hardware and software configurations.

- **OS:** Ubuntu 22.04.4 LTS x86_64
- **CPU:** AMD Ryzen Threadripper PRO 5965WX
- **GPU:** NVIDIA RTX A6000 × 2
- **Memory:** 256GB
- **Model:** gpt-4o-2024-08-06 (training data up to Oct 2023)

4.2 RQ1: Effectiveness of CrossProbe

Approaches for comparison. Besides the Basic CoT approach introduced in our motivational study, we also include two state-of-the-art LLM-based deep learning library testing approaches, FuzzGPT [8] and YanHui [13], for a more comprehensive comparison. It is worth mentioning that FuzzGPT and YanHui are originally designed for different purposes: FuzzGPT for edge-case fuzzing and YanHui for detecting model optimization bugs. In our evaluation, we adapt them to compare with CrossProbe from the following two perspectives.

- ***Quality of code transfer***. FuzzGPT and YanHui are example-based approaches, which require raw issue examples and instructive prompts. Comparatively, CrossProbe is a knowledge-based approach, leveraging API usages and seed issues in the source framework. We first compare the correctness of the code generated by all approaches to investigate which approach can achieve higher quality of code transfer.
- ***Performance in bug detection***. FuzzGPT and YanHui are essentially fuzzing approaches that focus on covering unexplored input combinations, which may not necessarily trigger bugs (i.e., besides bug detection, test coverage is also an important concern for the two approaches). In contrast, CrossProbe is a more directed bug detection approach that targets at validating whether bugs observed in a source framework also affect the target framework rather than performing comprehensive explorations (i.e., coverage is not a primary concern in our work). Therefore, we do not compare test coverage of different approaches but directly compare the detected bugs and check their overlaps.

For fair comparisons, we control the amount of input tokens (for cost concerns) and data source for different approaches. For FuzzGPT, we randomly select four samples (two from PyTorch and two from TensorFlow) to construct a prompt. For YanHui, we extract the error description, and choose 2-4 similar samples from the same framework for a round of generation. The average tokens and the corresponding cost on GPT-4o are listed in Table 3.

Table 3. Average Tokens and Cost for Different Approaches

Approach	Knowledge	Code	Total Input	Output	Cost (USD)
Basic CoT	35	4,760	4,795	4,733	$6.88
FuzzGPT	104	4,760	4,864	3,925	$5.96
YanHui	1,125	4,622	5,747	4,381	$6.75
CrossProbe	2,848	2,298	5,146	3,044	$5.02

Knowledge: CoT steps, few-shot examples of issue-triggering code, documentation. **Code:** pure code snippets from issues.

Evaluation metrics. We adopt three metrics to evaluate the performance of all compared approaches. The first two metrics, introduced in Section 2.2, assess the quality of code transfer: *1) syntactically valid outputs* and *2) functionality-preserving outputs*. The third metric, *real bugs*, evaluates the ability to detect bugs. During testing, we collect test cases that result in runtime errors or assertion failures and manually inspect these errors to identify genuine bugs.

Table 4. Generation Quality with Different Approaches

Approach	Outputs	Syntactically Valid	Functionality Preserving	Real Bugs
Basic CoT	116	18 (15.5%)	11 (9.5%)	0
FuzzGPT	116	55 (47.4%)	24 (20.6%)	5
YanHui	116	49 (42.2%)	16 (13.8%)	2
CrossProbe	116	65 (56.0%)	49 (42.2%)	19

Results. To ensure fair comparisons, we conduct the same iterations of generation (116) for all of the approaches. Table 4 presents the quality of the test cases generated with different approaches. Compared with other approaches, CrossProbe has significant improvements in generating functionality-preserving outputs, and can detect more real bugs.
Illustrative example #1. We present an example bug exclusively detected by CrossProbe to demonstrate its capability of transferring knowledge via leveraging mirroring issues.

```
1 conv = nn.Conv1d(1, 65537, 3, padding=1)
2 x = torch.ones([1, 1, 3])
3 y_mps = conv.to("mps")(x.to("mps"))
```

Listing 2. Special Configuration in PyTorch #129207

In Listing 2, the code can trigger an expected output when the argument of `Tensor.to` is set to `"mps"`. This API is a configuration method, which influences the program behaviors later. We find that LLM-based fuzzing approaches such as FuzzGPT and YanHui are unlikely to modify the tensors by adding the call of “`to()`” API. Their mutation points usually lie in `nn.Conv1d`, because it is the core API in the code snippet. As a result, the generated code may contain multiple variants of the arguments in the core API, leaving the statement of tensor (Line 3) unmodified. Therefore, they may not be effective to cover the side effects of previous configurations and generate corresponding test cases. For CrossProbe, since the related issue has been reported in PyTorch, it can directly learn the case of “convolution on Apple GPU” and transfer the testing point to TensorFlow. The transferred test case is valid in TensorFlow, and triggers an analogous issue, which has been reported to developers (TensorFlow #71577).

```
keras.mixed_precision.set_global_policy("mixed_float16")
model.compile(
    loss="sparse_categorical_crossentropy",
    optimizer=keras.optimizers.RMSprop())
```

Listing 3. Special Input Arguments in TensorFlow #66374

Illustrative example #2. In Listing 3, the code can trigger a runtime error with mixed floating-point data and autograph compilation. The argument in the `set_global_policy` and `compile` method include a string. For such an argument, the documentation may not list all the possible cases. Instead, the strings are described as *the name* of data types or functions. It requires additional knowledge to find out correct choices for testing. Currently, FuzzGPT and YanHui lack the capability to effectively leverage the full documentation of APIs and modules across different frameworks, such as constructing a full list of "names of mixed precision" or "names of loss function". Therefore, they may miss some input cases during fuzzing. For CrossProbe, these cases can be retrieved from existing issues in PyTorch (#46807). Given the testing point of "mixed float16" and "cross entropy", the generated code can directly explore the specific input combination and target at testing the analogous scenario in TensorFlow (#66374).

Analysis of results. As discussed in Section 2.2, the ratio of syntactically valid outputs evaluates the ability of an approach to correctly use APIs in code generation. In our results, the Basic CoT approach is left behind because it does not effectively leverage any API knowledge. The prompts of YanHui provide some information on the data and type of the API during the concentration process, but the use of knowledge is restricted to a single framework. FuzzGPT can achieve higher performance because the prompts provide examples from both the source and target frameworks, which can serve as useful references when transferring code. Compared to the above approaches, the prompts of CrossProbe provide further information and knowledge about the source and target frameworks, clearly pointing out the API pairs and usages. Therefore, CrossProbe behaves the best in generating valid code.

On the other hand, the ratio of functionality-preserving outputs measures the ability of an approach to choose appropriate APIs in code generation. In our results, CrossProbe shows significant improvements compared with other approaches. Such a great performance can be attributed to two reasons. First, CrossProbe has better input seeds. The screening process filters out the code that is not suitable for transfer. As a result, CrossProbe would not attempt to convert the code involving APIs without analogous ones in the target framework. Second, CrossProbe has more extensive knowledge of APIs. The distinction process provides information on which APIs are functionality-equivalent and how to use them. Therefore, CrossProbe can perform a more accurate code transfer from the source framework to the target framework.

Answer to RQ1. Compared with other LLM-based approaches, CrossProbe can transfer test cases with a significantly higher success rate. The high ratios of syntactically valid and functionality-preserving outputs can substantially reduce the effort in manual validation of the generated test cases for finding real bugs. The test cases generated via knowledge transfer can also effectively explore testing points that are hard to cover with fuzzing approaches.

4.3 RQ2: Ablation Study

To figure out how the three step of CrossProbe contribute to the code transfer, we conduct ablation studies by removing the key information for each step.

Influence of alignment. Table 5 presents the quality of generated code and detected bugs with or without the BERT-based API matching model. The differences in the output code and the detected bugs show that, having a higher amount of matched APIs can improve the performance of code transfer and bug detection. Without the API matching model, fewer API pairs are identified, which negatively impacts the subsequent screening and distinction processes. For example, when analyzing the results, we find that a number of suitable seed issues are discarded because the API pairs are missing from the database. Due to this reason, there are fewer test cases generated, and fewer APIs are tested. As a result, the number of detected bugs also decreases.

Table 5. The Performance of CrossProbe with or without Matching Model

Setting	Matched APIs	Inputs	Outputs	Functionality-Preserving	Bugs
With Matching Model	427	182	116	49 (26.9% of inputs)	19
Without Matching Model	298	182	72	26 (14.3% of inputs)	6

Influence of screening. Table 6 presents the quality of generated code and detected bugs with or without the process of screening. The comparison shows that the input without screening mainly affects the ratio of syntactically valid code generated by LLMs. Without screening, the input source code may contain APIs that do not have the functionality-equivalent counterparts in the target framework. As discussed in Section 2.2, the gap for functionalities causes a significant drop in the success rate of generating syntactically valid code. Regarding the filtered input cases obtained during the screening process, 10.7% are false positives, which in fact could help produce a valid output. However, these outputs are not functionality-preserving after further inspection. Overall, the screening process is effective in identifying useful inputs.

Table 6. The Performance of CrossProbe with or without Screening

Setting	Inputs	Outputs	Syntactically Valid	Valid in Filtered
With Screening	182	116	65 (56.0%)	N/A
Without Screening	182	182	**72 (39.6%)**	**7 (10.7%)**

Table 7. CrossProbe with or without Distinction

Setting	Outputs	Functionality-Preserving	Wrong Args	Empty	Untargeted
With Distinction	116	49 (42.2%)	22	9	36
Without Distinction	116	**28 (24.1%)**	39	14	35

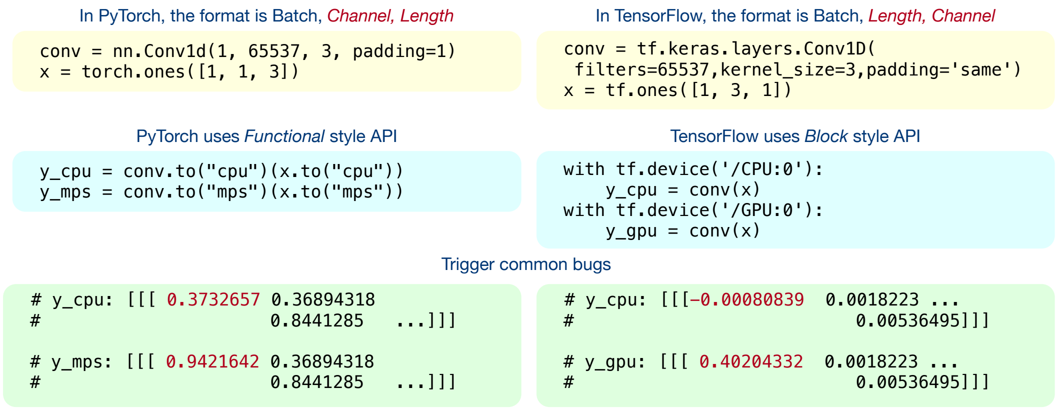
Influence of Distinction. Table 7 presents the experimental results with and without distinction prompts. Without the distinction process, the ratio of functionality-preserving outputs will drop significantly for the following reasons. First, the LLM lacks sufficient knowledge about the usage difference between APIs, and the generated code may not fully conform to the correct API usage of the target framework. This results in an increase in the number of "Wrong Args" errors. Second, the usage information of the target API is not comprehensive, and in such cases LLMs can only infer the API usage based on their own training data. As a result, low-quality code such as empty or incomplete statements could be generated.

Answer to RQ2. The alignment, screening, and distinction processes in CrossProbe all contribute to the improvement in the quality of code transfer and the number of detected bugs. The API matching model mainly confines the search space of code generation. The screening and distinction processes mainly help improve the correctness of the generated output. CrossProbe reaches its best performance with all the processes enabled.

4.4 RQ3: Usefulness

Bugs detected by CrossProbe. To evaluate the usefulness of CrossProbe, we review its generated code and categorize the code that can trigger the following errors.

- **Assertion errors** that arise when the assertions in the test cases are violated. CrossProbe has detected 13 such issues and the reasons include:
  - 8 operations on CPU and GPU produce different results, such as Conv1D on MPS (TensorFlow #71577).
  - 3 model conversions violate the expected changes, such as ONNX model export (PyTorch #136860).
  - 2 operators produce wrong results for specific inputs, such as softmax (PyTorch #123911).
- **Crash errors** that cause abrupt program termination when executing the generated test cases. CrossProbe has detected 6 such issues and the reasons include:
  - 4 environment issues when the code fails to run on specific platforms. For example, C++ library link errors, such as Metal backend (TensorFlow #62383).
  - 2 unsupported data types or operations that the framework fails to handle, such as adding int to tuple (PyTorch #136837).


In PyTorch, the format is Batch, *Channel, Length*
In TensorFlow, the format is Batch, *Length, Channel*
PyTorch uses *Functional* style API
TensorFlow uses *Block* style API
Trigger common bugs

```
conv = nn.Conv1d(1, 65537, 3, padding=1)
x = torch.ones([1, 1, 3])
```

```
conv = tf.keras.layers.Conv1D(
 filters=65537,kernel_size=3,padding='same')
x = tf.ones([1, 3, 1])
```

```
y_cpu = conv.to("cpu")(x.to("cpu"))
y_mps = conv.to("mps")(x.to("mps"))
```

```
with tf.device('/CPU:0'):
    y_cpu = conv(x)
with tf.device('/GPU:0'):
    y_gpu = conv(x)
```

```
# y_cpu: [[[ 0.3732657 0.36894318
#                       0.8441285   ...]]]

# y_mps: [[[ 0.9421642 0.36894318
#                       0.8441285   ...]]]
```

```
# y_cpu: [[[-0.00080839  0.0018223 ...
#                          0.00536495]]]

# y_gpu: [[[ 0.40204332  0.0018223 ...
#                          0.00536495]]]
```

Fig. 6. An Example of Different Results on CPU and GPU

To demonstrate the usefulness of CrossProbe in transferring code and finding bugs, we present two real bugs found by CrossProbe below.

Example bug #1. Figure 6 shows an example of different results produced by CPU and the Metal backend of GPU. The code is transferred from PyTorch (#129207). In this test case, a layer of Conv1D is applied to a tensor. The code implementation for PyTorch and TensorFlow is mainly different in the following aspects.

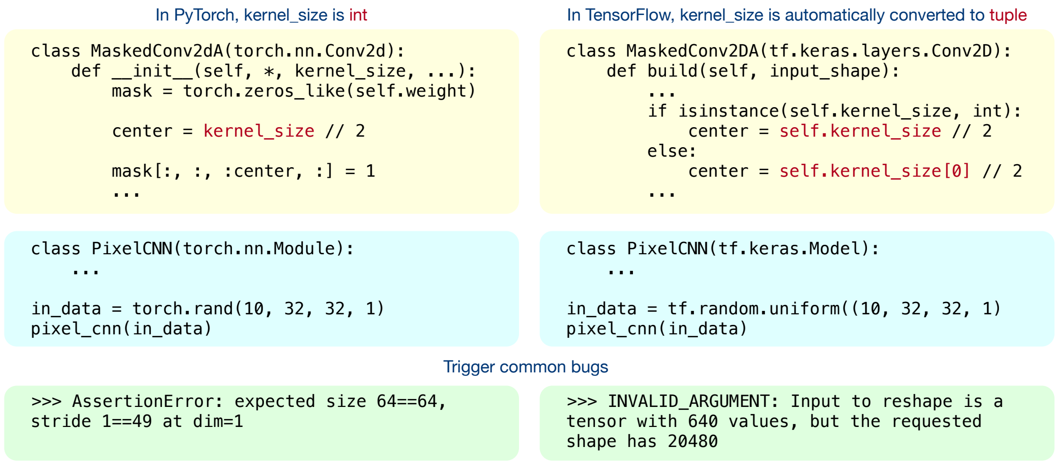
- **Tensor shape.** In PyTorch, the shape list represents the batch size (N), the number of channels (C), and the length of signal sequence (L), respectively. However, in TensorFlow, the shape is recognized by the batch shape, the steps, and channels, where the last two parameters are swapped.
- **Operation API style.** In PyTorch, GPU operations are triggered by calling functional APIs, and the usage is to convert the tensor to the target device. In contrast, TensorFlow uses block to determine the computing device. GPU operations should be wrapped into the `with` scope.

The above differences make it challenging for LLMs to transfer the code. For example, the code generated by YanHui fails to use the `with` block for the device, and the computation is not tested on the GPU. The code generated by the Basic CoT approach produces the wrong shape and fails to swap the channel and length parameters. In CrossProbe, the prompts include the API usage information, as illustrated in Listing 4. The explanation of each parameter is listed in detail, which effectively enables CrossProbe to follow the API usage to generate correct code.

```
# tf.nn.Conv1d
# Input: (N, C, L) or (C, L), where N is a batch size,
#        C denotes a number of channels, L is a length of signal sequence.
...
# tf.keras.layers.Conv1D
# data_format: If you never set it, then it will be "channels_last".
# Input shape: If data_format="channels_last":
#               A 3D tensor with shape: (batch_shape, steps, channels)
```

Listing 4. Usage Provided by the Distinction Process

With the successfully transferred code, the analogous problem of the unexpected output is also triggered in TensorFlow. This common issue is revealed because both PyTorch and TensorFlow are relying on the same dependency, GPU backend. In TensorFlow, the computation errors on Metal have not been noticed yet, and the problem cannot be easily detected by fuzzing the TensorFlow framework. CrossProbe takes advantage of the knowledge from PyTorch. Since PyTorch users first reported this problem, we can transfer the code and confirm an untested issue in TensorFlow. This bug is confirmed and discussed in TensorFlow #71577, and has been fixed in the latest version of macOS Sequoia 15.1.


In PyTorch, kernel_size is int
In TensorFlow, kernel_size is automatically converted to tuple
Trigger common bugs

```
class MaskedConv2dA(torch.nn.Conv2d):
    def __init__(self, *, kernel_size, ...):
        mask = torch.zeros_like(self.weight)

        center = kernel_size // 2

        mask[:, :, :center, :] = 1
        ...
```

```
class MaskedConv2DA(tf.keras.layers.Conv2D):
    def build(self, input_shape):
        ...
        if isinstance(self.kernel_size, int):
            center = self.kernel_size // 2
        else:
            center = self.kernel_size[0] // 2
        ...
```

```
class PixelCNN(torch.nn.Module):
    ...

in_data = torch.rand(10, 32, 32, 1)
pixel_cnn(in_data)
```

```
class PixelCNN(tf.keras.Model):
    ...

in_data = tf.random.uniform((10, 32, 32, 1)
pixel_cnn(in_data)
```

```
>>> AssertionError: expected size 64==64,
stride 1==49 at dim=1
```

```
>>> INVALID_ARGUMENT: Input to reshape is a
tensor with 640 values, but the requested
shape has 20480
```

Fig. 7. An Example of Unexpected Shape Error

Example bug #2. Figure 7 shows another example of mismatched shape size of deep learning models. The issue is originally reported in PyTorch (#136837), which is triggered by writing custom layers based on Conv2D. Similarly, without proper guidance, LLMs may make mistakes on the code transfer. In TensorFlow, the layer can be created with a kernel size of integer type by assuming the shape as a square. However, after the layer is constructed, the `self.kernel_size` is automatically converted to a tuple type. In our experiments, we observe that both FuzzGPT and YanHui fail in adapting the access of tuple type. In contrast, CrossProbe is aware of the usage of `kernel_size` by referring to the documentation, which mentions that the parameter can be “int or tuple/list of 2 integer”. Therefore, the correct code can be generated, and the internal bugs will be tested.

The code transferred to TensorFlow can also trigger errors about the tensor shape, which involves complex operations on layers and data. This case aligns with our finding of *common mistakes* mentioned in Section 1. In this example, the deep learning model is highly customized in PyTorch. It is challenging for LLM-based fuzzing approaches to cover such a case with only the knowledge from TensorFlow. CrossProbe leverages the knowledge from PyTorch and can successfully detect this error by constructing analogous customized model in TensorFlow.

Answer to RQ3. CrossProbe is useful for finding bugs in real-world frameworks by transferring issues from other frameworks. The errors exposed by the transferred code can be difficult to trigger when only leveraging single-framework data (e.g., historical issues) during testing. CrossProbe demonstrates promising results in generating test cases for undiscovered problems and has successfully detected 19 assertion failures or crashes in PyTorch and TensorFlow.

5 Discussion

5.1 Feedback From the Community

Through our responsible disclosure process, all bugs identified by CrossProbe have been submitted to the respective GitHub repository of PyTorch and TensorFlow. These bug reports have yielded concrete feedback from the community:

Timely fixes and integration of tests. TensorFlow #71577 is resolved by Apple in macOS Sequoia 15.1. PyTorch #145735 and #145203 are successfully patched in PR #146085 with test cases integrated into PyTorch's official unit testing suite.

Upstream dependency resolution. In five cases where the root cause resided in platform implementations, two CUDA-related issues (TensorFlow #87432, #83379) are acknowledged by NVIDIA and awaiting further fix. For three Apple MPS backend issues, TensorFlow #71577 is resolved as mentioned above, and TensorFlow #62859, PyTorch #144824 are being diagnosed by Apple.

Under further discussion. Certain issues have revealed differences in the API behavior between PyTorch and TensorFlow. For example, the input of TensorFlow #71638 may cause a negative output in TensorFlow, but a crash in PyTorch. These cases have initiated technical discussions among the developers of the frameworks, which are still ongoing at the time when we submit the paper.

In summary, among the 19 issues reported by us, 21% are fully resolved and 56% are in active remediation phases.

5.2 Comparison with Fuzzing

The experimental results in Section 4 demonstrate that targeted testing via knowledge transfer (CrossProbe) and existing fuzzing approaches (FuzzGPT and YanHui) exhibit distinct strengths in detecting bugs in deep learning systems. This section further analyzes the fundamental differences between two different types of approaches and proposes to synergize them to achieve a comprehensive testing of deep learning frameworks.

Characteristics of fuzzing approaches. Fuzzing approaches, through mutated input generation, excel at discovering crashes and output inconsistencies inside deep learning frameworks[8, 13]. Their strength lies in exploring unexpected input combinations that developers might overlook. However, as illustrated in Section 4.2, their strategies may fail to trigger side effect issues of configurations and miss some input cases during mutation.

Advantages of knowledge transfer. Our knowledge-based approach addresses some of the missing cases precisely by leveraging known problematic patterns from historical issues of analogous frameworks. By systematically transferring and adapting test cases (Section 3), we achieve targeted validation of framework behaviors.

Rather than viewing the two types of approaches as competing alternatives, we can consider them as complementary components in a comprehensive testing strategy. Fuzzing serves as a broad-spectrum detector for generic inconsistencies and is particularly effective to explore unknown failure patterns. On the other hand, knowledge transfer acts as a precise and targeted tool for auditing framework-specific behaviors and is especially valuable for mature systems with accumulated historical bug patterns.

5.3 Threats to Validity and Limitations

The validity of our study is subject to certain threats, primarily related to the stability of the LLM outputs and the constraints of our testing environment for the generated code. One key threat to validity is the inherent unpredictability of outputs from GPT-4o, a black-box online service that does not allow for precise control over generated content, resulting in potential variations in responses across identical prompts. This lack of control can introduce inconsistencies that affect the reliability of our results.

Currently, CrossProbe supports PyTorch and TensorFlow. With the growing popularity of LLMs, emerging frameworks such as Apple's MLX and LLM-specific solutions like vLLM have introduced new runtime environments. To address this, we plan to extend CrossProbe 's compatibility to these frameworks in future work. The adaptation effort is lightweight, primarily involving API labeling and description mapping to ensure consistent behavior across frameworks, without requiring large-scale model re-training or architectural changes.

Additionally, our testing environment is limited to GPUs from Apple and NVIDIA. This restriction imposes compatibility constraints, as some code snippets may be designed to leverage platform-specific libraries such as AMD's ROCm or Intel's MKL-DNN, which are unsupported in our setup. As a result, any code dependent on these alternative libraries is likely to encounter execution failures within our testing environment. To mitigate these limitations, we identify and exclude incompatible code during the screening phase, as described in Section 3.3. This step ensures that our study remains focused on code suitable for transfer within the capabilities of our testing environment, thereby improving the robustness of our evaluation process.

6 Related Work

Bugs in deep learning libraries. Researchers have conducted extensive investigations into bugs and testing within deep learning libraries. To characterize these bugs, a substantial body of literature [3–5, 14, 17, 34] has examined the symptoms and root causes of various types of bugs in deep learning frameworks, including common programming errors, performance issues, deployment challenges, compiler bugs, and the newly emerging class of optimization bugs.
Traditional approaches to testing deep learning libraries. To test libraries using input data, Dwarakanath et al. [11] employed metamorphic testing [33], where training and testing data are mutated. Potential bugs in the library implementations are identified when violations of metamorphic relations are observed during testing.

For testing deep learning models, CRADLE [28] introduced an effective test oracle for deep learning libraries, utilizing differential testing [23]. Framework bugs are exposed when inconsistencies are found in model outputs. Building on this, Audee [15] focused on fuzz testing Deep Neural Network (DNN) models. Audee uses popular DNN models as seeds and mutates inputs and weights to generate new models for testing. In addition, DeepMutation++[16] adopts a different strategy by categorizing DNNs into Feed-Forward Neural Networks (FNNs) and Recurrent Neural Networks (RNNs), applying specific mutation strategies for each type based on their structural differences. LEMON[41] presents an approach for generating models by employing heuristic rules to guide mutations, thereby increasing the likelihood of exposing bugs. NNSmith [21] leverages SMT solvers to generate DNN models that adhere to the constraints of the computation graph.

Beyond input and model testing, API testing is also a crucial aspect. FreeFuzz [42] collects API calls from open-source projects, identifies parameters in function calls, and mutates these parameters to generate new test cases for deep learning libraries. DocTer [46] aims to fuzz API functions by extracting documentation from deep learning libraries. The extracted documentation is normalized, parsed into dependency trees, and used to generate test functions that satisfy the constraints of the APIs.

LLM-empowered approaches to testing deep learning libraries. With the advent of LLMs, researchers have begun leveraging these models for software testing. TitanFuzz [7] was the first to introduce LLMs to fuzzing deep learning libraries. It generates seed inputs using prompts describing the API and mutates these inputs by masking portions of the code, asking LLMs to fill in the missing parts. FuzzGPT [8] employs few-shot prompting, combining examples of code and descriptions to generate code for library testing.

Besides fuzzing the whole framework for detecting general bugs, the fine-grained scope of LLM-based testing has also been studied recently. YanHui [13] focuses on detecting model optimization bugs using a prompt design paradigm called "concentration and diffusion", which incorporates detailed domain knowledge of model optimization.

7 Conclusion

In conclusion, this paper introduces an innovative approach to enhance test generation for deep learning (DL) frameworks by leveraging analogous bugs across frameworks, termed as "mirroring issues". Our tool, CrossProbe, utilizes large language models to effectively learn from existing issues in one framework and generate targeted test cases for another framework, thus achieving cross-framework bug detection. Through the introduction of three key processes - *alignment*, *screening*, and *distinction* - we address the challenges of incompatible functionalities and divergent implementations between frameworks, which often complicate test case generation. The experimental results on PyTorch and TensorFlow, two most popular DL frameworks, show that CrossProbe significantly improves the efficiency of code generation and achieves a much higher success rate in transferring issues compared to two state-of-the-art LLM-based testing methods. Our findings demonstrate the potential of CrossProbe in cross-framework bug detection, paving the way for more robust DL frameworks and contributing to a more reliable and trustworthy ecosystem of DL applications.

8 Data Availability

We open-source CrossProbe and the associated knowledge database of API on [6], to facilitate further research of the code transfer between deep learning frameworks.

Acknowledgment

We thank our anonymous reviewers for their constructive comments and insightful suggestions on the paper. This work is partially supported by the National Natural Science Foundation of China (Grant No. 62372219), the Australian Research Council Discovery Projects (DP230101196, DP240103068), and the National Key Research and Development Program of China (Grant No. 2019YFE0198100).

References

[1] Meta AI. 2023. Llama 2. https://ai.meta.com/llama/. Accessed: 2023-09-22.

[2] Mihalj Bakator and Dragica Radosav. 2018. Deep learning and medical diagnosis: A review of literature. *Multimodal Technologies and Interaction* 2, 3 (2018), 47.

[3] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, and Xin Peng. 2021. Characterizing Performance Bugs in Deep Learning Systems. *CoRR* abs/2112.01771 (2021). arXiv:2112.01771 https://arxiv.org/abs/2112.01771

[4] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* (mar 2023). doi:10.1145/3587155 Just Accepted.

[5] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of deep learning based mobile applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 674–685.

[6] CrossProbe2024. 2025. *CrossProbe2024/CrossProbe: 2025-04-12*. doi:10.5281/zenodo.15201811

[7] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. arXiv:2212.14834 [cs.SE]

[8] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. arXiv:2304.02014 [cs.SE]

[9] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Automated Relational API Inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. doi:10.1145/3540250.3549085

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. doi:10.18653/v1/n19-1423

[11] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 118–128. doi:10.1145/3213846.3213858

[12] Zhipeng Gao. 2020. When deep learning meets smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1400–1402.

[13] Hao Guan, Guangdong Bai, and Yepang Liu. 2024. Large Language Models Can Connect the Dots: Exploring Model Optimization Bugs with Domain Knowledge-Aware Prompts. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1579–1591. doi:10.1145/3650212.3680383

[14] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A comprehensive study of real-world bugs in machine learning model optimization. In *The 45th International Conference on Software Engineering (ICSE)*. 147–158.

[15] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 486–498.

[16] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1158–1161. doi:10.1109/ASE.2019.00126

[17] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside tensorflow. In *International Conference on Database Systems for Advanced Applications*. Springer, 604–620.

[18] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[19] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599* (2023).

[20] Geert Litjens, Clara I Sánchez, Nadya Timofeeva, Meyke Hermsen, Iris Nagtegaal, Iringo Kovacs, Christina Hulsbergen-Van De Kaa, Peter Bult, Bram Van Ginneken, and Jeroen Van Der Laak. 2016. Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis. *Scientific reports* 6, 1 (2016), 1–11.

[21] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. doi:10.1145/3575693.

3575707

[22] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).

[23] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[24] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, Vol. 2024.

[25] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A multilanguage static analysis of python programs with native C extensions. In *International Static Analysis Symposium*. Springer, 323–345.

[26] Bence Nagy, Tibor Brunner, and Zoltán Porkoláb. 2021. Unambiguity of Python Language Elements for Static Analysis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 70–75.

[27] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[28] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. doi:10.1109/ICSE.2019.00107

[29] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[30] Sebastian Ramos, Stefan Gehrig, Peter Pinggera, Uwe Franke, and Carsten Rother. 2017. Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling. In *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1025–1032.

[31] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI EA '21)*. Association for Computing Machinery, New York, NY, USA, Article 314, 7 pages. doi:10.1145/3411763.3451760

[32] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[33] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[34] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 968–980.

[35] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, Miles McCain, Alex Newhouse, Jason Blazakis, Kris McGuffie, and Jasmine Wang. 2019. Release Strategies and the Social Impacts of Language Models. arXiv:1908.09203 [cs.CL]

[36] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 120–131.

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[38] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[39] 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 *EMNLP*.

[40] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

[41] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 788–799.

[42] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[44] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 385, 22 pages. doi:10.1145/3491102.3517582

[45] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[46] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–188.

[47] Chuan Yan, Mark Huasong Meng, Fuman Xie, and Guangdong Bai. 2024. Investigating Documented Privacy Changes in Android OS. *Proc. ACM Softw. Eng.* 1, FSE, Article 119 (July 2024), 24 pages. doi:10.1145/3660826

[48] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 627–638.

[49] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 359–371.

Received 2024-10-31; accepted 2025-03-31