

The Seeds of the FUTURE Sprout from History: Fuzzing for Unveiling Vulnerabilities in Prospective Deep-Learning Libraries

Zhiyuan Li^{†‡}, Jingzheng Wu^{†§¶}, Xiang Ling^{†§¶}, Tianyue Luo[†], Zhiqing Rui^{†‡}, Yanjun Wu^{†§¶}

[†]Institute of Software Chinese Academy of Sciences

[‡]University of Chinese Academy of Sciences

[§]Key Laboratory of System Software (Chinese Academy of Sciences)

[¶]State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences

Abstract—The widespread application of large language models (LLMs) underscores the importance of deep learning (DL) technologies that rely on foundational DL libraries such as PyTorch and TensorFlow. Despite their robust features, these libraries face challenges with scalability and adaptation to rapid advancements in the LLM community. In response, tech giants like Apple and Huawei are developing their own DL libraries to enhance performance, increase scalability, and safeguard intellectual property. Ensuring the security of these libraries is crucial, with fuzzing being a vital solution. However, existing fuzzing frameworks struggle with target flexibility, effectively testing bug-prone API sequences, and leveraging the limited available information in new libraries. To address these limitations, we propose FUTURE, the first universal fuzzing framework tailored for newly introduced and prospective DL libraries. FUTURE leverages historical bug information from existing libraries and fine-tunes LLMs for specialized code generation. This strategy helps identify bugs in new libraries and uses insights from these libraries to enhance security in existing ones, creating a cycle from history to future and back. To evaluate FUTURE's effectiveness, we conduct comprehensive evaluations on three newly introduced DL libraries. Evaluation results demonstrate that FUTURE significantly outperforms existing fuzzers in bug detection, success rate of bug reproduction, validity rate of code generation, and API coverage. Notably, FUTURE has detected 148 bugs across 452 targeted APIs, including 142 previously unknown bugs. Among these, 10 have been assigned CVE IDs. Additionally, FUTURE detects 7 bugs in PyTorch, demonstrating its ability to enhance security in existing libraries in reverse.

Index Terms—Fuzzing, DL Libraries, Historical Bug.

I. INTRODUCTION

Artificial intelligence (AI) continues to lead technological innovation, with large language models (LLMs) rapidly gaining widespread application [1]. Within the pipeline of AI, deep learning (DL) has made significant strides [2], prompting a growing demand for efficient and versatile programming frameworks. This demand has led to the development of DL libraries (e.g., PyTorch [3] and TensorFlow [4]). Despite their comprehensive capabilities, these libraries face limitations in scalability, flexibility, and proprietary control, which are critical for tech giants like Apple and Huawei. To address these

limitations and meet the complex requirements of LLMs, these companies invest heavily in developing their own DL libraries.

Emerging DL libraries, including Apple MLX [5], Huawei MindSpore [6], and OneFlow [7], aim to improve performance through enhanced computational efficiency and advanced features. However, their complexity may introduce bugs that pose critical risks in sectors like healthcare [8], finance [9], and autonomous driving [10]. Identifying and addressing bugs in these libraries is essential for the safety of downstream DL systems. Despite high level of attention and adoption these libraries have received in their early development stages, there is a noticeable lack of mature and comprehensive testing methodologies. This gap underscores the need to ensure their security and reliability.

Fuzzing has proven highly effective in detecting software bugs by automatically generating test cases that expose unexpected behaviors [11]. Existing fuzzing methods for DL libraries are mainly categorized into API-level fuzzing [12]–[15] and model-level fuzzing [16]–[18]. API-level fuzzing focuses on individual API functions to uncover bugs triggered by anomalous API inputs but may miss complex bugs due to its inability to construct intricate API sequences [19]. Model-level fuzzing tests entire models against inputs that exploit architectural or weight bugs. Although it addresses many limitations of API-level fuzzing, model-level fuzzing covers a limited range of APIs [20]. Recent works (e.g., TitanFuzz [21] and FuzzGPT [22]) leverage LLMs to address some limitations of API-level fuzzing methods. While API-level, model-level, and recent LLM-based methods have demonstrated excellent performance in detecting bugs within DL libraries, applying these methods to newly introduced and prospective DL libraries poses several challenges:

C1: Lack of Target Flexibility. Existing fuzzing methods, primarily designed for PyTorch and TensorFlow, lack the flexibility needed to adapt to new libraries. Applying these methods to new libraries requires extensive resource collection and code modifications, which are both time-consuming and labor-intensive. Despite this fundamental limitation, other challenges persist within the existing methods, as outlined below.

[¶]Jingzheng Wu and Xiang Ling are the corresponding authors.

C2: Lack of Bug-prone API Sequences. Previous API-level methods [12]–[15] struggle to effectively test API sequences. TitanFuzz [21] leverages LLMs to address this limitation. However, in essence, TitanFuzz’s generation and mutation still rely on the inherent capabilities of LLMs, resulting in the random generation of multi-API code snippets. This approach fails to test bug-prone API sequences, leading to an excessively large search space and limited efficiency in finding bugs.

C3: Limited Available Information. Existing methods utilize documentation [12], open-source code snippets [13], and historical bugs [22] to guide the fuzzing process. However, newly introduced and prospective libraries typically only provide documentation, sometimes including code examples, in their early stages. The availability of open-source code snippets and historical bugs is extremely limited. Therefore, documentation becomes the primary resource, but existing methods that rely on it, such as Docter [12], are limited to testing individual APIs, as mentioned in C2.

C4: Lag in Knowledge of LLMs. Despite the rapid updates in LLMs, it typically takes several months for the latest models to incorporate knowledge about newly introduced libraries. As a result, LLM-based fuzzers (*e.g.*, TitanFuzz [21], FuzzGPT [22] and Fuzz4All [23]) that rely on pre-trained models are significantly constrained in their effectiveness.

To overcome these challenges, we propose FUTURE, the first universal fuzzing framework tailored for both newly introduced and prospective DL libraries. Existing libraries contain a wealth of historical bug information, including complex and bug-prone API sequences that closely mimic real-world usage scenarios. The basic idea of FUTURE is to leverage the historical bug information from existing libraries (referred to as source libraries) to identify bugs in newly introduced and prospective libraries (referred to as target libraries). Firstly, we design a label-specific GitHub spider to crawl issues related to bugs in source libraries. We extract codes that reproduce these issues as historical bug codes (see Section III-B). To bridge the gap between source and target libraries, we utilize a universal prompt template to leverage the limited information available in the API documentation and code examples of the target libraries. Using prompts crafted from this template, we invoke LLMs to generate code pairs that illustrate how source and target libraries implement the same API functions. FUTURE then mutates these code pairs to construct datasets for fine-tuning. With the fine-tuned LLMs, we generate seed codes by converting historical bug codes from source to target libraries and generating codes that invoke the API of the target libraries (see Section III-C). Finally, we conduct differential testing on three newly introduced libraries to demonstrate the effectiveness of FUTURE. We identify bugs, extract bug-prone API inputs, and test these inputs on the source libraries to unveil undetected bugs (see Section III-D). In summary, FUTURE makes the following contributions:

- **Universal Fuzzing Framework:** We propose FUTURE, the first universal fuzzing framework for both newly introduced and prospective DL libraries. This framework significantly reduces the effort required to adapt fuzzing

techniques to any new DL library, making it a forward-thinking tool. Additionally, FUTURE is the first fuzzing method that targets Apple MLX.

- **From History to FUTURE and Back:** We establish a code conversion mapping between existing and prospective libraries by fine-tuning LLMs. This mapping allows us to convert historical bug codes from existing libraries into seed codes for prospective libraries, pioneering a fuzzing method from historical insights to future anticipations. Furthermore, by leveraging bugs found in prospective libraries, FUTURE enhances security in existing libraries, completing a cycle from future back to history.
- **Bug Detection:** FUTURE demonstrates remarkable efficacy by detecting 148 bugs across 452 targeted APIs in Apple MLX, Huawei MindSpore, and OneFlow, including 142 previously unknown bugs. Among these bugs, 10 have been assigned CVE IDs. In addition, FUTURE has identified 7 bugs in PyTorch. We release the details of detected bugs and FUTURE implementation at <https://github.com/Redmpt1on/FUTURE>.

II. BACKGROUND AND RELATED WORK

A. Deep learning and DL libraries

In the burgeoning field of AI, DL represents a paradigmatic shift, employing models that emulate the intricate neural structures of the human brain [24]. This advanced branch of machine learning (ML) relies on neural networks to identify subtle patterns in large datasets, supporting decision-making and eliminating the need for manual feature extraction [25].

DL libraries serve as pivotal components within DL systems, providing tools and interfaces for designing, implementing, and efficiently training neural models. They simplify complex mathematical operations and hardware interactions, making deep learning more accessible to developers and researchers [26]–[30]. Early libraries like Theano [31] and Caffe [32] established the foundation, while PyTorch [3] and TensorFlow [4] revolutionized the landscape with features like automatic differentiation [33] and adaptive computational graphs [34], accelerating research and model refinement.

Recent developments include libraries like Apple MLX [5], Huawei MindSpore [6] and OneFlow [7], which optimize performance and scalability for DL tasks. These libraries enhance computational efficiency in large-scale and distributed environments, addressing the needs of an era marked by exponentially growing data volumes, model complexities, and the development of LLMs. The widespread deployment of these libraries provides essential infrastructure for developing DL models across various sectors including healthcare [8], autonomous vehicles [35] and finance [9]. Consequently, it is vital to maintain rigorous oversight to ensure these libraries meet high standards of quality and security.

B. Fuzzing with Large Language Models

The emergence of LLMs has transformed various domains, providing unprecedented capabilities in text generation [36], code generation [37], and more [38]–[41]. These capabilities

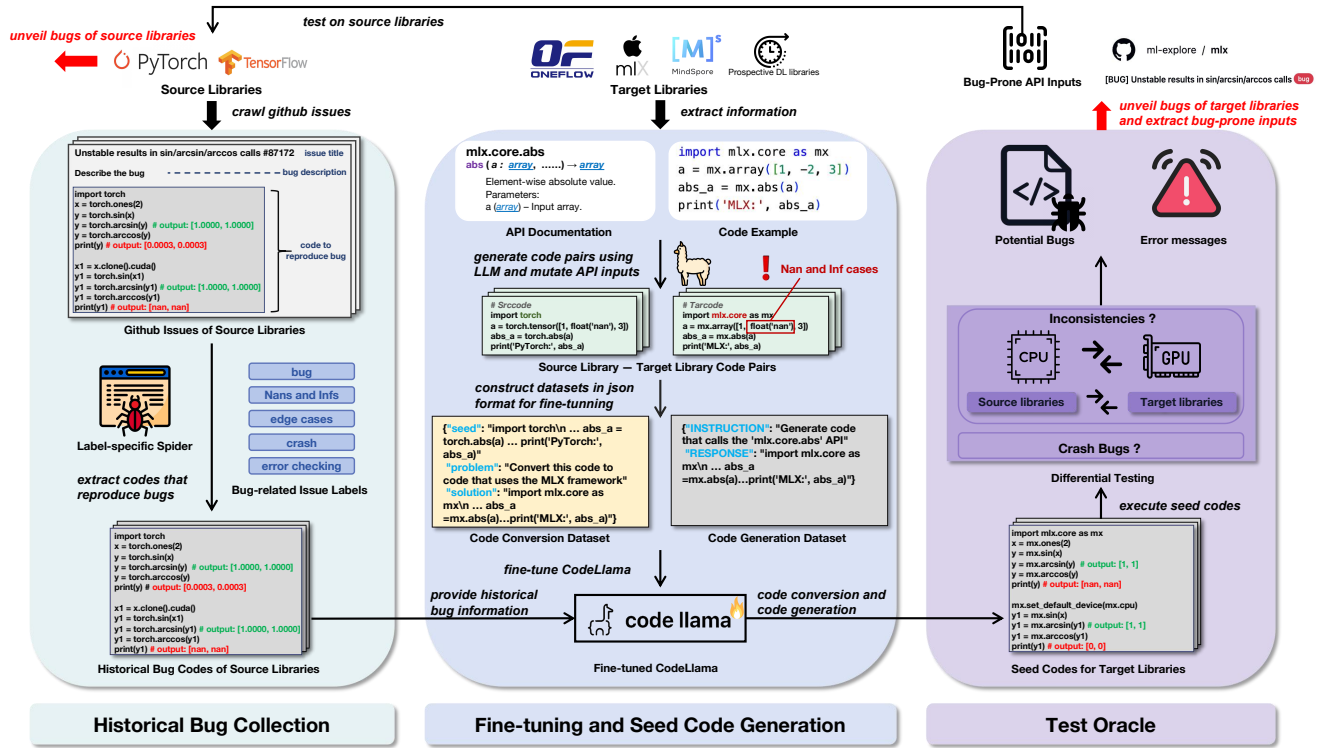


Fig. 1: **Overview of FUTURE.** FUTURE leverages historical bug information from source libraries and available API information from target libraries to realize code pairs generation, dataset construction, LLM fine-tuning, and seed code generation. Utilizing these seed codes, FUTURE unveils bugs in target libraries through test oracle. Insights gained from these bugs are used to enhance the security of the source libraries, completing a cycle from history to future and back.

are revolutionizing fields such as content creation [42], conversational AI [43], and software testing [44].

Pre-trained LLMs (e.g., GPT-3 [45] and Llama [46]) serve as the foundational models, offering broad capabilities across diverse linguistic tasks due to their extensive training on text data. General-purpose fine-tuned LLMs (e.g., Codex [47] and CodeLlama [48]) extend these models by focusing on specific capabilities like coding generation, making them versatile tools in various applications [49]. However, they sometimes lack the nuanced understanding required for highly specialized tasks.

Task-specialized fine-tuned models such as CodeLlama-Python and CodeLlama-Instruct [50] address this limitation by tailoring general-purpose fine-tuned models to excel in particular domains. These models integrate domain-specific knowledge during further fine-tuning, enabling them to solve problems that pre-trained and general-purpose fine-tuned models struggle with, such as understanding complex domain jargon or predicting highly specialized outcomes in fields like medical diagnostics or financial forecasting.

Integrating LLMs into the fuzzing process allows fuzzers to explore systems in ways traditional methods cannot, investigate potential failure modes, and ensure robustness against a wider range of input scenarios [51]. LLMs are extensively employed in fuzzing frameworks for compilers, protocols, and various other domains [52]–[54]. Several frameworks have

utilized LLMs to enhance DL library fuzzing. TitanFuzz [21] is the first approach that directly leverages a general-purpose fine-tuned LLMs (Codex and InCoder) to generate and mutate DL programs for fuzzing. However, due to limitations in model capability and knowledge lag, TitanFuzz cannot effectively test complex, bug-prone API sequences or provide immediate testing for newly introduced libraries. FuzzGPT [22] demonstrates that LLMs can be prompted or fine-tuned to resemble historical bug-triggering programs. However, FuzzGPT is limited by its reliance on historical bug-triggering programs from the libraries being tested, which is impractical for newly introduced and prospective libraries. Fuzz4All [23] effectively utilizes LLMs for input generation and mutation across various software systems under test (SUTs), however, when applied to DL libraries, it encounters the same challenges as TitanFuzz.

III. APPROACH

A. Overview

As shown in Fig. 1, FUTURE comprises three main phases: historical bug collection, fine-tuning and seed code generation, and test oracle.

In the historical bug collection phase (Section III-B), we focus on existing libraries. To gather historical bug information, we design a label-specific spider to crawl bug-related

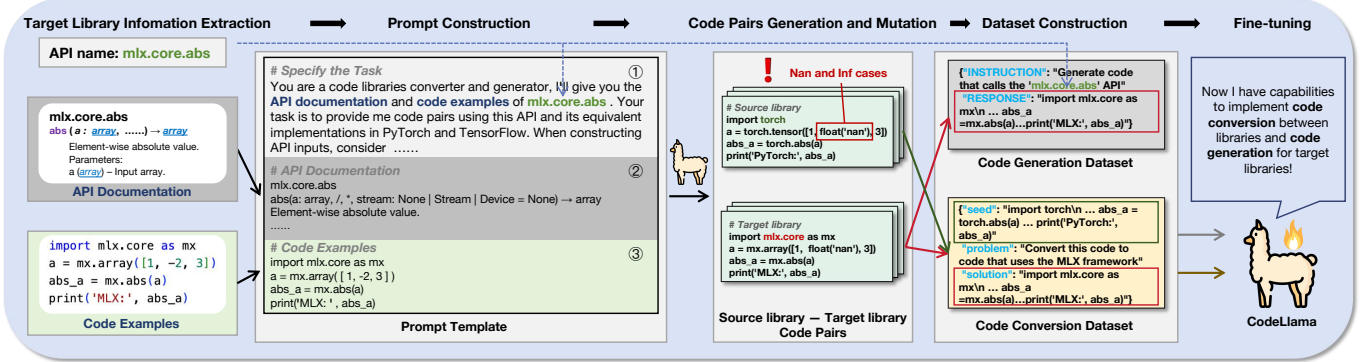


Fig. 2: **Datasets Construction and Fine-tuning.** In the prompt template, we emphasize the importance of constructing API inputs that consider NaNs and Infs, edge cases, and scenarios likely to trigger API error checking and crashes. Due to space constraints, these details are omitted in the figure.

GitHub issues, retrieving code snippets that trigger bugs. These snippets serve as historical bug codes.

In the fine-tuning and seed code generation phase (Section III-C), we concentrate on newly introduced and prospective DL libraries. We first collect API documentation and code examples (Section III-C1). Then, we construct a prompt template to leverage these resources (Section III-C2). By invoking LLMs with prompts, we generate and mutate code pairs that consist of an API call in a target library and its corresponding implementation in a source library (Section III-C3). The obtained code pairs are then converted into datasets to fine-tune LLMs (Section III-C4). With the fine-tuned LLMs, we convert the historical bug codes into codes using the target libraries. These converted codes, along with codes randomly generated by fine-tuned LLMs for the target libraries, form the seed codes (Section III-C5).

In the test oracle phase (Section III-D), we execute the seed codes to perform differential testing on the target libraries, identifying potential bugs through abnormal behaviors such as crashes and inconsistencies. Utilizing the bugs detected by FUTURE in the target libraries, we extract API inputs that are likely to trigger bugs and attempt to detect unaddressed bugs in the source libraries.

B. Historical Bug Collection

Several existing DL libraries (e.g., PyTorch [3] and TensorFlow [4]) have significant user engagement, resulting in thousands of bug reports. Users typically interact with project developers via GitHub issues, reporting potential bugs. These reports include basic descriptions, system environment information, and code snippets to reproduce the issues. Developers address these issues by committing fixes for verified bugs and labeling them accordingly.

Inspired by this process, to collect historical bug codes, we design a fully automated, label-specific spider. This spider parses issue pages, categorizes issues by labels, retrieves issue contents, extracts code snippets and perform preprocessing.

Specifically, our spider traverses each page of issues, retrieving the title and ID of each entry. It subsequently accesses the

issue pages to extract code snippets tagged with keywords such as “Standalone code to reproduce the issue”, “Usage example” or “Code example”. These extracted snippets are then processed to enhance usability by importing necessary dependencies and removing non-code elements. These snippets are then saved, with directory and file names generated based on the issue’s title and ID. To ensure valid filenames, special characters are replaced, and a numerical suffix is appended in cases of duplicate names. The saved snippets serve as FUTURE’s historical bug codes, enabling systematic organization and storage for further analysis and examination.

C. Fine-tuning and Seed Code Generation

1) *Target Library Information Extraction:* Firstly, FUTURE extracts API documentation and code examples from the official documentation of target libraries. For each API, the API name *Name*, API documentation *Doc*, and code examples *Code* are saved in a triplet format $\text{APLinfo} = \{Name, Doc, Code\}$. For APIs lacking documentation and referencing other APIs, FUTURE extracts information from the referenced APIs to ensure comprehensive coverage.

2) *Prompt Construction:* After retrieving APLinfos, we devise a universal prompt template to construct prompts. The constructed prompts consist of three parts:

Specify the Task. This part defines the task to obtain code pairs that consist of an API call in a target library and its corresponding implementation in a source library, more details of this part are shown in Fig. 2 (①).

API Documentation. For APIs with extensive documentation, we streamline the prompts by removing redundant information to stay within the *max_tokens* limit of LLMs.

Code Examples. For code examples that present multiple usage methods for a single API, we extract different methods. By deconstructing the code examples, we derive new prompts P' from the original prompts P . To be more specific, we first use regular expressions [55] to determine whether the *Name* appears multiple times in the code example. If the *Name* appears only once, we directly use the code example to construct the prompt (i.e., $p' = p$). If the *Name* appears

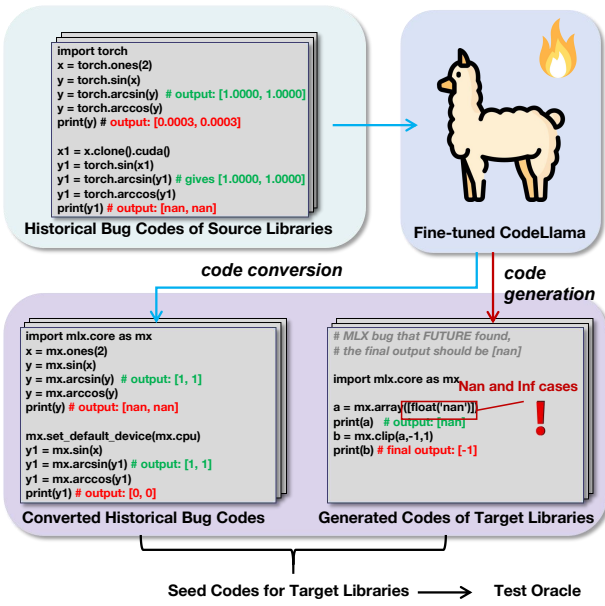


Fig. 3: **Seed Code Generation.** With the task-specialized fine-tuned LLMs, we perform code conversion and code generation to obtain the seed codes for test oracle.

multiple times, we use abstract syntax trees [56] to decompose and reassemble the code example, splitting k usage methods into multiple separate code examples, thereby decomposing p into $p' = \{p_1, p_2, \dots, p_k\}$, where $p \in P$ and $p' \subseteq P'$. Fig. 2 (②,③) illustrates the incorporation of API documentation and code examples into our prompt template.

3) *Generation and Mutation of Code Pairs:* Subsequently, we query LLMs with new prompts P' to obtain code pairs $CP = \{CP_1, CP_2, \dots, CP_n\}$, where n represents the number of prompts eventually constructed. Each code pair $CP_i = (S_i, T_i)$, $i \in (1, 2, 3, \dots, n)$ in CP consists of source library code S and corresponding target library code snippets T . To manage costs, we limit the number of code pairs generated per API. We then randomly select variables in the API input data and mutate them to random numbers. Assuming that each code pair CP_i undergoes m mutations, $n * m$ mutated code pairs are acquired, which can be represented as $CP' = \{CP_{11}, CP_{12}, \dots, CP_{1m}, \dots, CP_{nm}\}$. By deconstructing code examples, generating code pairs and mutating them, we acquire multiple code pairs for a single API, resulting in more robust data for fine-tuning.

4) *Dataset Construction:* After obtaining CP' , we convert them into datasets for fine-tuning, resulting in two datasets:

Code Generation Dataset: Since LLMs are not pre-trained with knowledge of prospective libraries, our goal is to equip them to generate code for the target libraries. Therefore, we construct the code generation dataset using T as the “RESPONSE”, providing standard answers for code generation, as illustrated in Fig. 2.

Code Conversion Dataset: To equip LLMs with the capability to convert code snippets from the source libraries to target

libraries, we construct a triplet format dataset using CP' . This dataset designates S as “seed” and T as “solution”, with the “problem” explicitly stated as “Convert this code to code that uses the target library (MLX/MindSpore/OneFlow)”.

5) *Fine-tuning and Seed Code Generation:* We design a universal fine-tuning template to enable users to adapt FUTURE to their own requirements with minimal effort. There are three popular fine-tuning techniques for pre-trained models [57]: Fine Tuning [58], Parameter-efficient Fine-Tuning (PEFT) [59], and Prompt Tuning [60]. We employ the Low-Rank Adaptation (LoRA) [61] from PEFT in the template.

Consider $W \in \mathbb{R}^{d \times k}$ as the weight matrix of LLMs. Instead of updating W directly, LoRA introduces two low-rank matrices $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$, where $r \ll \min(d, k)$. The weight update is then parameterized as $\Delta W = AB$. During fine-tuning, the effective weight matrix W' is given by:

$$W' = W + \Delta W = W + AB$$

The low-rank matrices A and B are updated during the fine-tuning process, while the original weight matrix W remains fixed. The update rule for A and B typically follows the gradients of the loss function L with respect to these matrices. Let $\theta = \{A, B\}$ denote the parameters of the low-rank matrices. The gradient descent update step for θ is given by:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} L$$

where η is the learning rate. By leveraging the low-rank structure, we reduce the number of trainable parameters and thus decrease the computational cost and memory overhead, while still enabling effective fine-tuning of the model.

Using this fine-tuning template, we fine-tune three separate models: using only the code generation dataset, only the code conversion dataset, and both datasets together. By fine-tuning on these datasets, the LLMs not only learn various code mappings between the source libraries and target libraries but also acquire knowledge of the target libraries.

The task-specialized fine-tuned LLMs are then employed for seed code generation, detailed in Fig. 3. On one hand, FUTURE leverages the code conversion capability of the fine-tuned LLMs to convert historical bug codes His into potential bug codes Pot for the target libraries. In the Pot , there is a rich and diverse collection of complex and bug-prone API sequences that closely mimic real-world usage scenarios. On the other hand, FUTURE utilizes the code generation capability to generate diverse code snippets Gen that call APIs of the target libraries. Together, Pot and Gen form the seed codes. During the acquisition of Pot and Gen , we perform automated preprocessing same as Section III-B. Even though His has been processed earlier, the inherent uncertainty of LLMs still necessitate this preprocessing to ensure the usability of the seed codes.

This strategy effectively transcends the limitations posed by relying solely on historical bug codes from the source libraries. It allows FUTURE to utilize historical bug information from source libraries, which TitanFuzz [21] cannot do. Additionally, it facilitates the generation of code snippets of target libraries,

TABLE I: GitHub issues and bug codes collected.

Source library	Issue Label	Issue Number	Bug Codes
PyTorch	bug	315	178
	Nans and Infs	138	43
	edge cases	184	84
	error checking	275	86
	crash	388	139
Tensorflow	bug	9975	4503
ToTal	-	11275	5033

even when models incorporating knowledge of these libraries are unavailable—something TitanFuzz is unable to achieve.

D. Test Oracle

After obtaining seed codes, we implement differential testing, focusing on two critical aspects:

Crash Bugs. During the execution of the seed codes, we monitor for crashes such as aborts, segmentation faults, runtime errors, and floating point exceptions. Bugs exposed by such crashes may further trigger security vulnerabilities.

Inconsistencies. We scrutinize the execution results across different computational backends (CPU/GPU) for inconsistencies, which often indicate underlying bugs. Additionally, since each fuzzing seed code corresponds to an implementation in the source libraries, following [21], we calculate the Euclidean distance [62] to measure the discrepancies between results. We only consider discrepancies exceeding a predefined threshold T as “potential bugs”. Only when these discrepancies are reported to and confirmed by developers do we classify them as triggered bugs caused by inconsistencies between the source libraries (Src libs) and target libraries (Tar libs).

After conducting the test oracle, we report and perform a statistical analysis of the detected bugs. This analysis allows us to summarize and extract bug-prone API inputs, which are then used to automatically test the APIs of the source libraries.

In FUTURE, there are only two manually involved parts. The first is extracting bug-prone API inputs during the bug reporting process. The second involves manually inspecting the causes of failures in code pairs generation, which constitute a small portion. This inspection helps us identify documentation issues related to certain APIs. Apart from these two parts, FUTURE is fully automated.

IV. EVALUATION

A. Implementation

DL Library Selection. We select PyTorch (v2.0.0) [3] and TensorFlow (v2.13.0) [4] as the source libraries for FUTURE due to their extensive application. FUTURE can be applied to any newly introduced or prospective DL library. To assess its effectiveness, we focus on three target libraries introduced in recent years: Apple MLX (v0.0.10), Huawei MindSpore (v2.2.13), and OneFlow (v0.9.1). These open-source libraries represent the cutting-edge in DL library development, making them ideal candidates for evaluating FUTURE’s effectiveness.

Historical Bug Collection. FUTURE targets all bug-related issues up to March 20, 2024, processing a total of 11,275 issues from PyTorch and TensorFlow on GitHub. From these, we extract 5,033 valid historical bug codes, as detailed in Table I. Due to constraints in computational resources and time, we use the most recent 1,000 TensorFlow bug codes and all available PyTorch bug codes, resulting in a total of 1,530 historical bug codes utilized in this study.

Large Language Models. FUTURE utilizes CodeLlama-13B [48] for generating code pairs and fine-tuning, we select CodeLlama due to its advanced performance in open models and large input context support. The foundation model and its weights are sourced from Hugging Face [63], providing a cost-effective solution to users as it is available for free.

Fine-tuning Datasets. We generate code pairs for 128, 156, and 168 APIs for MLX, MindSpore, and OneFlow respectively. For each API, we generate 5 code pairs, which are then mutated 100 times, resulting in a fine-tuning dataset of over six hundred thousand entries.

Fine-tuning Setups. During fine-tuning, we quantize the model’s parameters to 4 bits. For the LoRA configuration, we follow the official tutorial provided by PEFT [59]. As for the training parameters, the learning rate η is set at 3e-4 and the *max_steps* is set to 400. We allocate one-tenth of the training datasets for validation, conducting validations every 20 steps.

B. Experimental Setup

Baselines. Since the target libraries of FUTURE are newly introduced, most existing fuzzing methods are not readily applicable to all of them. As FUTURE is an API-level fuzzer, we focus on state-of-the-art API-level fuzzers such as FreeFuzz [13], DeepREL [64], NablaFuzz [14], TensorScope [15], and TitanFuzz [21], while also considering model-level fuzzers like Muffin [18] for a comprehensive understanding. Among these, FreeFuzz and DeepREL require extensive data collection and code modifications, making them resource-intensive for our needs. Muffin, limited to backends compatible with Keras, is not directly applicable to our target libraries without significant manual adaptations. Therefore, we select TitanFuzz, NablaFuzz and TensorScope as our baselines and adapt them to test our target libraries. In addition, to achieve a more comprehensive comparison, we selected few-shot learning [65] as the fourth baseline to verify the effectiveness and rationality of fine-tuning in FUTURE through comparison.

All baseline fuzzers are modified minimally to ensure compatibility with our target libraries. We make several main adaptations as below: (1) replacing TitanFuzz’s deprecated model with gpt-3.5-turbo, (2) updating API lists to match our targeted APIs, (3) modifying codes to save generated snippets.

In few-shot, for a fair comparison, we choose few-shot + CodeLlama (w/o fine-tuning) and we adopt chain-of-thought (CoT) prompting. For different tasks, we provide specific task descriptions along with 10-shot examples from our conversion or generation dataset, respectively. Using this context, we perform code conversion and generation accordingly.

TABLE II: **Summary of CVEs detected by FUTURE.** All detected CVEs are found in OneFlow and due to the lack of strict parameter checking in APIs, indicating that OneFlow urgently needs to optimize the parameter validation mechanisms to prevent crashes or inconsistencies that could be exploited by attackers.

CVE ID	Symptom	Vulnerable API
CVE-2024-36730	Crash	zeros/ones/new_ones/empty
CVE-2024-36732	Crash	tensordot
CVE-2024-36734	Crash	var
CVE-2024-36735	Src libs/Tar libs	eye
CVE-2024-36736	Src libs/Tar libs	permute
CVE-2024-36737	Crash	full
CVE-2024-36740	Crash	scatter/scatter_add
CVE-2024-36742	Crash	scatter_nd
CVE-2024-36743	Crash	dot
CVE-2024-36745	Crash	index_select

TABLE III: Summary of bug detection on target libraries.

	Total	Confirmed		Won't fix
		Unknown	Known	
MLX	35	32	0	3
MindSpore	30	22	6	2
OneFlow	83	83	0	0
Total	148	137	6	5

Environment. We use an Ubuntu 20.04 server equipped with an Intel Xeon Gold 6130 CPU and a V100-32GB GPU.

C. Metrics

Number of Bugs. We report potential bugs detected by FUTURE to developers via GitHub. We count only those bugs labeled as “bug” as detected by FUTURE.

Success rate of bug reproduction. We define Suc as converted codes that successfully reproduce the behaviors (error messages or outputs in bug reports) observed in the original bug codes. The success rate quantifies the effectiveness of FUTURE in code conversion and is calculated as Success Rate = $\frac{N_{Suc}}{N_{His}}$, where N represents the number of codes.

Validity rate. For code generation, the validity rate is calculated as the ratio of the number of unique valid codes Val to the total number of generated codes All . A code is valid if it executes without exceptions and invokes the target API at least once. The validity rate is given as Validity Rate = $\frac{N_{Val}}{N_{All}}$.

API Coverage. We measure API coverage by calculating the proportion of the APIs utilized in Suc and Val , relative to the total number of targeted APIs. Specifically, APIs that appear in Suc are denoted as $SucAPI$, and APIs that appear in Val are denoted as $ValAPI$. The API coverage is given by:

$$API\ Coverage = \frac{(N_{SucAPI} \cup N_{ValAPI}) \cap N_{TarAPI}}{N_{TarAPI}}$$

where N_{TarAPI} represents the total number of targeted APIs.

D. Research Questions

To assess the effectiveness of FUTURE, we conduct studies answering the following research questions:

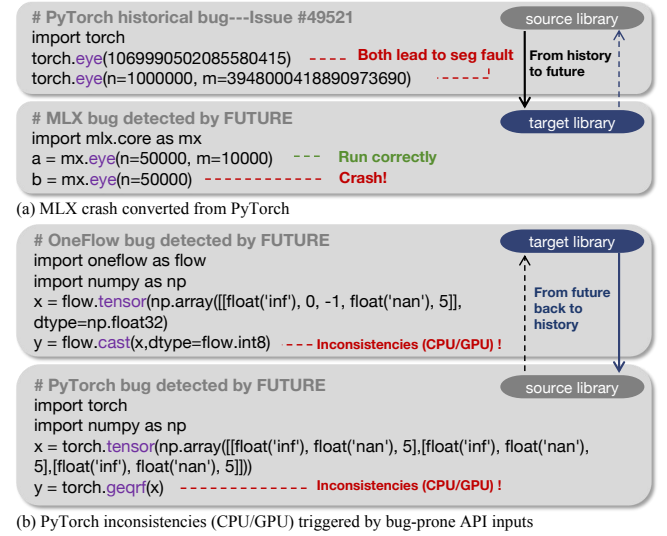


Fig. 4: **Example bugs found by FUTURE.** We provide two bug examples to illustrate that FUTURE not only uses historical bug information from source libraries to unveil bugs in target libraries but also leverages bugs found in target libraries to identify bugs that still reside in source libraries.

- **RQ1:** Can FUTURE detect real-world bugs in newly-introduced DL libraries?
- **RQ2:** How do key components and different settings of FUTURE influence its effectiveness?
- **RQ3:** What is the fuzzing performance of FUTURE compared to the state-of-the-art techniques?

V. RESULT ANALYSIS

A. RQ1: Bug Detection

We first investigate the effectiveness of FUTURE in bug detection. FUTURE detects 148 bugs across Apple MLX, Huawei MindSpore, and OneFlow, with 142 confirmed as previously unknown. Among these bugs, 10 have been assigned CVE IDs as detailed in Table II. Our submissions on GitHub are recognized with five “good first issue” labels by MLX developers. Additionally, we identify 7 bugs in PyTorch. Since PyTorch is not one of our target libraries, we do not provide a detailed analysis of these bugs here. Statistics of bugs on the target libraries are presented in Table III.

Fig. 4(a) shows a historical bug code from PyTorch. In this bug, excessively large parameters passed to `torch.eye` trigger a segmentation fault. FUTURE converts this code into code snippets using target libraries. We find that on MLX, `mlx.core.eye` operates correctly when both parameters are set to large values. However, setting only one parameter to a large value triggers a crash. This issue is labeled as a bug and immediately fixed by MLX developers. This demonstrates that FUTURE can utilize historical bug information from existing libraries to unveil bugs in prospective libraries.

Fig. 4(b) illustrates a bug detected by FUTURE through its capability to generate random codes for target libraries. We find that arrays containing NaNs and Infs can trigger many

TABLE IV: Causes of bug detected on target libraries.

	Total	EC	NI	LD	DBI	MPC
MLX	35	4	17	3	4	7
MindSpore	30	0	25	0	1	4
OneFlow	83	0	51	3	1	28
Total	148	4	93	6	6	39

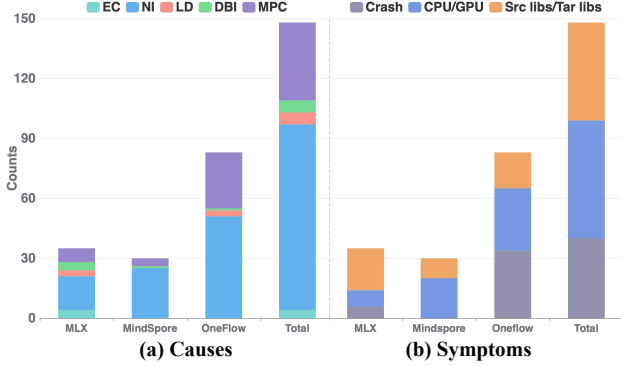


Fig. 5: Statistical distribution of causes and symptoms of bugs detected in target libraries using FUTURE.

bugs in target libraries. By extracting bug-prone API inputs and testing them on the APIs of source libraries, we identify that torch.geqr exhibits inconsistencies on CPU and GPU. This issue persists in the latest version of PyTorch and has been confirmed as a bug. This demonstrates that FUTURE uses insights from prospective libraries to enhance security in existing ones, creating a cycle from history to future and back.

Beyond bugs, 20 issues detected by FUTURE, though not classified as bugs, are recognized by developers as valuable enhancement issues. Additionally, FUTURE identifies 12 documentation problems, we report them and they have all been addressed in the latest versions of the official documentation.

Bug Cause Analysis. Based on labels of historical bugs and the actual circumstances of bugs detected by FUTURE, we classify the causes of the detected bugs into five categories:

- **Nans and Infs (NI):** Bugs in this category involve the presence of NaNs or Infs in two specific contexts: as variables in an input array or as API parameters.
- **Missing Parameter Checking (MPC):** These bugs result from insufficient or flawed parameter validation within APIs, allowing parameters that do not meet constraints to pass checks, leading to incorrect results or crashes.
- **Edge Cases (EC):** These bugs involve bugs in APIs when handling boundary values for certain data types.
- **Different Backend Implementations (DBI):** Bugs in this category arise from discrepancies in the implementation of DL libraries across different backends (CPU/GPU), affecting the outputs under identical inputs and parameters.
- **Logic Deficiency (LD):** Some APIs suffer from logic deficiencies, preventing them from functioning as intended.

We categorize the bugs detected by FUTURE according

TABLE V: Symptoms of bug detected on target libraries.

	Total	Crash	CPU/GPU	Src libs/Tar libs
MLX	35	6	8	21
MindSpore	30	0	20	10
OneFlow	83	34	31	18
Total	148	40	59	49

to these causes and compile statistics. Table IV details the number of bugs per category, and Fig. 5(a) illustrates the distribution of each cause within the bugs detected in each target library. For bugs involving inputs or parameters with NaNs and Infs, and where outputs vary across different backends, we categorize the cause as “NI”. Only when a bug does not involve NaNs and Infs and there are discrepancies in outputs across different backends do we classify it as “DBI”. Similarly, when NaNs and Infs used as parameters trigger a bug, we prioritize classifying it under “NI” rather than “MPC”.

Among all bugs, “NI” is the most common cause, accounting for 62.83% (93 out of 148). “MPC” is the second most common cause, accounting for 26.3% (39 out of 148), suggesting stricter parameter checking is needed during the development of DL libraries.

Bug Symptom Analysis. We also categorize the symptoms of the bugs based on the design of test oracle. Table V and Fig. 5(b) illustrate the distribution of bugs according to symptom categories, divided into three types:

- **Crash:** This category includes scenarios where seed code execution results in aborts, segmentation faults, runtime errors, floating point exceptions, and excessively long execution times without producing results or errors.
- **Inconsistencies (CPU/GPU):** These bugs are characterized by different outcomes for the same API executed on different backends, despite identical inputs and parameters.
- **Inconsistencies (Src libs/Tar libs):** These bugs are identified when the outcomes of the target library seed codes are consistent across different backends but differ from those of corresponding implementations in the source libraries, given the same inputs and parameters.

Our findings reveal that OneFlow has a higher incidence of bugs manifesting as crashes, accounting for 40.96% (34 out of 83) of the total bugs detected in OneFlow by FUTURE. This is likely due to inadequate handling of invalid inputs or parameters in OneFlow’s source code, often leading to program terminations. Additionally, FUTURE finds no crash-related bugs in MindSpore. We attribute this to MindSpore’s implementation of stricter parameter checks and more robust error handling mechanisms.

Answer to RQ1: FUTURE detects 148 bugs across MLX, MindSpore, and OneFlow, with 142 confirmed as previously unknown. The analysis on bug causes and symptoms provides valuable insights for further research in DL libraries.

B. RQ2: Impact of Key Components and Different Settings

To study how each component of FUTURE contributes to the overall effectiveness, we conduct experiments based on the

TABLE VI: **Evaluation of historical bug collection.** In the brackets following the percentages, the numerator and denominator for the success rate are represented by N_{Suc} and N_{His} respectively; for API coverage, they are covered API and all targeted API. The detailed definitions of these metrics can be found in Section IV-C. The column **Bug detected** shows the number of bugs detected by FUTURE using historical bug codes with specific labels.

Historical Bug Codes on Source Libraries		Converted Codes on Target Libraries						
Source library	Issue Label	Success Rate	API Coverage			Bug detected		
			MLX	MindSpore	OneFlow	MLX	MindSpore	OneFlow
PyTorch	bug	11.8% (21/178)	17.2% (22/128)	9.0% (14/156)	11.3% (19/168)	2	1	5
	Nans and Infs	20.9% (9/43)	25.0% (32/128)	26.3% (41/156)	28.0% (47/168)	6	4	11
	edge cases	13.1% (11/84)	21.1% (27/128)	16.0% (25/156)	7.7% (13/168)	1	0	1
	error checking	15.1% (13/86)	8.6% (11/128)	12.2% (19/156)	14.9% (25/168)	0	1	1
	crash	11.5% (16/139)	13.3% (17/128)	21.8% (34/156)	18.5% (31/168)	2	0	2
TensorFlow	bug	8.6% (86/1000)	34.4% (44/128)	35.9% (56/156)	45.2% (76/168)	2	1	5
Total	-	10.2% (156/1530)	57.0% (73/128)	65.4% (102/156)	68.5% (115/168)	13	7	25

TABLE VII: Evaluation of different methods for constructing the fine-tuning dataset.

	Success Rate	Validity Rate	API Coverage			
			MLX	MindSpore	OneFlow	Total
FUTURE	10.2% (156/1530)	96.5% (2894/3000)	98.4% (126/128)	98.7% (154/156)	97.0% (163/168)	98.0% (443/452)
FUTURE-doc	3.4% (52/1530)	45.8% (1375/3000)	35.2% (45/128)	37.2% (58/156)	41.1% (69/168)	38.1% (172/452)
FUTURE-ex	4.4% (68/1530)	49.6% (1487/3000)	40.6% (52/128)	52.0% (81/156)	38.1% (64/168)	43.6% (197/452)
FUTURE-no	1.4% (21/1530)	36.1% (1083/3000)	28.9% (37/128)	28.2% (44/156)	19.6% (33/168)	25.2% (114/452)

¹ FUTURE represents fine-tuning with datasets constructed by generating and mutating code pairs. FUTURE-doc relies solely on API documentation and FUTURE-ex directly uses code examples to fine-tune CodeLlama. FUTURE-no represents the version without fine-tuning.

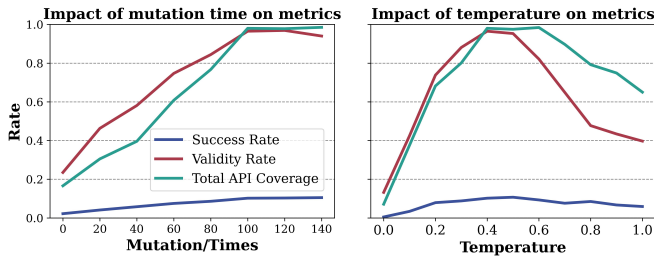


Fig. 6: Impact of various mutation times and temperature on success rate, validity rate, and total API coverage.

four key components: (1) historical bug collection, (2) dataset construction, (3) fine-tuning, and (4) seed code generation.

1) *Historical Bug Collection:* We perform code conversion on 1,530 historical bug codes mentioned in Section IV-A, with the detailed results shown in Table VI.

First, we observe that issues labeled “Nans and Infs” in PyTorch provide the most extensive API coverage and are instrumental in bug detection. This observation confirms our prompt design strategies as discussed in Section III-C2. Second, we note that the success rate of bug reproduction for TensorFlow’s historical bug codes is notably low at 8.6%. This is likely due to the vast number of TensorFlow APIs, many of which are undeveloped in the target libraries after conversion. Despite this, the historical bug codes of TensorFlow contribute significantly to overall API coverage and assist in detecting 8 bugs within three test libraries.

In summary, our experiments demonstrate that utilizing historical bug information from existing DL libraries contributes

to detecting bugs in prospective DL libraries. As the collection of bugs in existing DL libraries continues to accumulate, this contribution is expected to become increasingly effective. Among the 45 bugs detected by FUTURE using historical bug information, 33 involved multi-API sequences, proving that FUTURE effectively leverages bug-prone API sequences rather than randomly generating API sequences with LLMs, as seen in TitanFuzz.

2) *Dataset Construction:* Next, we examine how constructing and leveraging datasets for fine-tuning with different strategies influences the effectiveness of FUTURE.

How to Construct Datasets. We compare several variants of constructing datasets with different strategies. Table VII shows the performance metrics of these strategies.

We observe that constructing the fine-tuning dataset by generating and mutating code pairs significantly enhances the success rate, validity rate and API coverage compared to other strategies. Notably, the metrics of FUTURE-ex outperform FUTURE-doc, indirectly confirming that using code snippets as datasets more effectively simulates real-world code conversion and generation scenarios.

We evaluate the impact of varying mutation times on the performance metrics, shown in Fig. 6(a). We find that mutating code pairs significantly improves all metrics. However, when increasing mutations beyond 100, the benefits do not justify the additional time and resources needed. This finding guides us to cap mutations at 100 per code pair.

How to Leverage Datasets. To equip LLMs with the capabilities for code conversion and code generation, we construct two specialized datasets. We first investigate the impact of utilizing

TABLE VIII: The impact of different dataset combinations used for fine-tuning on FUTURE’s code generation effectiveness.

	Validity Rate	Total API Coverage
FUTURE	96.5% (2894/3000)	92.3% (417/452)
FUTURE-gen	81.1% (2432/3000)	83.8% (379/452)
FUTURE-conv	52.5% (1576/3000)	56.2% (254/452)
FUTURE-no	36.1% (1083/3000)	22.8% (103/452)

¹ **FUTURE** uses both datasets, **FUTURE-gen** performs fine-tuning with only code generation dataset, **FUTURE-conv** employs only code conversion dataset, and **FUTURE-no** indicates no fine-tuning at all.

TABLE IX: The impact of different dataset combinations on FUTURE’s code conversion effectiveness. All the variants of FUTURE remain consistent with the configurations in Table VIII.

	Success Rate	Total API Coverage
FUTURE	10.2% (156/1530)	94.5% (290/452)
FUTURE-gen	5.0% (77/1530)	32.7% (148/452)
FUTURE-conv	8.6% (132/1530)	52.0% (235/452)
FUTURE-no	1.4% (21/1530)	13.5% (61/452)

different combinations of these datasets on the effectiveness of FUTURE’s code generation capability. For each target library, we generate 1,000 random code snippets for evaluation.

Table VIII presents the impact of different dataset combinations on FUTURE’s code generation capability. **FUTURE-gen** achieves significantly higher validity rate and API coverage compared to **FUTURE-conv** and **FUTURE-no**, indicating that the code generation dataset substantially enhances FUTURE’s capability to generate relevant and effective code snippets.

Next, we conduct experiments to evaluate the impact of these different dataset combinations on FUTURE’s code conversion capability using the same four versions of FUTURE described above. We perform library conversion on the 1,530 historical bug codes mentioned in Section IV-A. The experimental results are detailed in Table IX. These results demonstrate that the code conversion dataset can enhance FUTURE’s success rate in converting code between libraries. Using both of the designed datasets maximizes their benefits, affirming their complementary nature.

3) *Fine-tuning*: We further assess the effectiveness of fine-tuning CodeLlama by comparing metrics with and without fine-tuning. Table VII presents the performance enhancements achieved through fine-tuning. The results show that **FUTURE** significantly outperforms **FUTURE-no**, with improvements exceeding threefold across all metrics, demonstrating the substantial benefits of further fine-tuning on general-purpose fine-tuned LLMs. Even with a model like CodeLlama-13B, which has limited performance, FUTURE still achieved notable results, highlighting the framework’s effectiveness regardless of the underlying model’s capabilities.

4) *Seed Code Generation*: The main goal of seed code generation is to implement code conversion and generation accurately. Therefore, we examine the impact of the *temperature* hyperparameter on FUTURE’s various performance. Fig. 6(b) shows the metrics of FUTURE at varying *temperature* settings. We observe that with the *temperature* set to the default

TABLE X: Comparison on API coverage.

	MLX	MindSpore	OneFlow	Total
FUTURE	98.4% (126/128)	98.7% (154/156)	97.0% (163/168)	98.0% (443/452)
TitanFuzz	29.7% (38/128)	89.1% (139/156)	82.9% (126/168)	67.0% (303/452)
NablaFuzz	-	-	78.0% (131/168)	28.9% (131/452)
TensorScope	-	30.1% (47/156)	-	10.4% (47/452)
Few-shot	59.4% (76/128)	77.56% (121/156)	81.5% (137/168)	73.9% (334/452)

TABLE XI: Comparison on code generation. In the table, for individual target library, the denominators for the percentages are all 1,000. They are omitted due to space limitations.

	MLX	MindSpore	OneFlow	Total
FUTURE	97.5%	95.2%	96.7%	96.5%
FUTURE-gen	83.4%	79.8%	80.0%	81.1%
FUTURE-conv	54.2%	44.3%	59.1%	52.5%
FUTURE-no	14.2%	45.3%	48.8%	36.1%
TitanFuzz	15.7%	41.5%	52.7%	36.6%
Few-shot	42.7%	70.5%	63.9%	59.0%

value of 0.4, FUTURE demonstrates superior capabilities in code conversion and code generation, effectively maximizing coverage across targeted APIs.

Answer to RQ2: The effectiveness of FUTURE is enhanced by its key components, which significantly boost success and validity rates, as well as API coverage and bug detection.

C. RQ3: Comparison with Other Work

In this section, we compare FUTURE with four baselines: TitanFuzz, NablaFuzz, TensorScope and few-shot learning.

We did not find any confirmed bugs using our baselines. Table X presents the API coverage of all evaluated baselines on our target libraries. NablaFuzz, which does not support MLX and MindSpore and relies on its proprietary database inaccessible to us, has its performance recorded only for OneFlow. Similarly, TensorScope’s design, which involves constraints among APIs of multiple libraries, makes migration to MLX and OneFlow challenging with minimal effort.

We observe that FUTURE achieves significantly higher API coverage across all target libraries, with an increase of 46.2% compared to TitanFuzz. TitanFuzz performs relatively well on MindSpore and OneFlow but poorly on MLX, likely due to the training data of gpt-3.5-turbo not including information of MLX. This highlights FUTURE’s suitability for prospective libraries, where existing fuzzers like TitanFuzz may be ineffective. Few-shot learning addresses this issue to some extent, but its performance still lags significantly behind fine-tuning. FUTURE achieves a 24.4% improvement on OneFlow compared to NablaFuzz and outperforms TensorScope on newer versions of MindSpore.

TitanFuzz first realizes a fully-automated framework to perform generation-based fuzzing directly leveraging LLMs. We conduct experiments comparing the code generation capability of different versions of FUTURE with TitanFuzz and few-shot learning. For each target library, we generate 1,000 code snippets. Table XI presents the validity rate of the snippets.

FUTURE significantly outperforms TitanFuzz in terms of validity rates across all target libraries. Specifically, the total

validity rate of FUTURE reaches 263.7% of TitanFuzz. For MLX, the validity rate is more than six times higher than TitanFuzz. *TitanFuzz's performance is only comparable to FUTURE-no*. These findings indicate that, with the FUTURE's code generation capability for newly introduced and prospective DL libraries significantly surpasses that of the state-of-the-art generation-based DL library fuzzing method.

Few-shot learning shows an improved validity rate across all libraries compared to TitanFuzz, but it still falls short of FUTURE. Through manual analysis of some of the code snippets generated by few-shot learning, we find that the quality of the generated code snippets deteriorate over time, with instances where the generated code even failed to adhere to the instructions in the initial prompt. In contrast, after fine-tuning, each generated code snippet is a new invocation of the fine-tuned model, which effectively avoids the issues encountered in few-shot learning.

Answer to RQ3: FUTURE consistently outperforms baselines in multiple metrics, showcasing its superior efficacy and adaptability for newly introduced and prospective libraries.

VI. THREATS TO VALIDITY

Internal: In our experiments, we employ CodeLlama to generate code pairs based on API documentation and code examples. Given the inherent uncertainties and variability in the performance of LLMs, not all generated code pairs may meet the desired quality standards. We mitigate this uncertainty through designing rigorous preprocessing and validity checks on the code pairs. By implementing these checks, we aim to reduce the impact of low-quality code pairs and maintain the reliability of our experimental results.

External: Applying FUTURE to DL libraries not initially targeted by the framework may compromise its effectiveness, particularly if users are unfamiliar with the fine-tuning process of LLMs. To address this threat, we provide a comprehensive fine-tuning template that guides users through the adaptation process. By offering this resource, we aim to empower users to effectively customize FUTURE to their specific needs with minimal effort, thereby enhancing its applicability and robustness across various DL libraries.

VII. CONCLUSION

In this work, we propose FUTURE, the first universal DL library fuzzing framework designed for both newly introduced and prospective DL libraries. More specifically, FUTURE collects historical bug codes from existing libraries, fine-tunes LLMs with limited available information. With the historical bug codes and fine-tuned LLMs, we generate seed codes and perform differential testing, enhancing security in both new and existing libraries. Our evaluation on three newly introduced libraries shows that FUTURE significantly outperforms existing fuzzers in multiple dimensions. Notably, FUTURE has detected 148 bugs across 452 targeted APIs, including 142 previously unknown bugs. Among these bugs, 10 have been assigned CVE IDs. Our submissions on GitHub

are recognized with five “good first issue” labels by MLX developers. Additionally, FUTURE detects 7 bugs in PyTorch, demonstrating the framework’s ability to utilize historical bug information to secure new libraries and enhance existing ones in reverse. In subsequent research, we aim to expand the range of FUTURE’s source libraries, incorporating a broader spectrum of historical bug information from various DL libraries. Additionally, we will implement more automated components to complete the cycle from future back to history.

ACKNOWLEDGMENT

We sincerely appreciate all the anonymous reviewers for their insightful and constructive comments to improve this work. This paper is supported by the the Strategic Priority Research Program of the Chinese Academy of Sciences under No. XDA0320401, the National Natural Science Foundation of China under No. 62202457, and the Open Source Community Software Bill of Materials (SBOM) Platform under No. E3GX310201. This paper is supported by YuanTu Large Research Infrastructure.

REFERENCES

- [1] E. Bonner, R. Lege, and E. Frazier, “Large language model-based artificial intelligence in the language classroom: Practical ideas for teaching,” *Teaching English with Technology*, 2023.
- [2] M. Soori, B. Arezoo, and R. Dastres, “Artificial intelligence, machine learning and deep learning in advanced robotics, a review,” *Cognitive Robotics*, 2023.
- [3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, 2019.
- [4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: a system for large-scale machine learning,” in *OSDI*, 2016.
- [5] A. Hannun, J. Digani, A. Katharopoulos, and R. Collobert, “MLX: Efficient and flexible machine learning on apple silicon,” 2023, <https://github.com/ml-explore>.
- [6] L. Huawei Technologies Co., “Huawei mindspore ai development framework,” in *Artificial Intelligence Technology*. Springer, 2022.
- [7] J. Yuan, X. Li, C. Cheng, J. Liu, R. Guo, S. Cai, C. Yao, F. Yang, X. Yi, C. Wu *et al.*, “Oneflow: Redesign the distributed deep learning framework from scratch,” *arXiv:2110.15032*, 2021.
- [8] A. Kleppe, O.-J. Skrede, S. De Raedt, K. Liestøl, D. J. Kerr, and H. E. Danielsen, “Designing deep learning studies in cancer diagnostics,” *Nature Reviews Cancer*, 2021.
- [9] M. Leo, S. Sharma, and K. Maddulety, “Machine learning in banking risk management: A literature review,” *Risks*, 2019.
- [10] A. Miglani and N. Kumar, “Deep learning models for traffic flow prediction in autonomous vehicles: A review, solutions, and challenges,” *Vehicular Communications*, 2019.
- [11] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE TSE*, 2019.
- [12] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, “Docter: Documentation-guided fuzzing for testing deep learning api functions,” in *IEEE ISSTA*, 2022.
- [13] A. Wei, Y. Deng, C. Yang, and L. Zhang, “Free lunch for testing: Fuzzing deep-learning libraries from open source,” in *IEEE/ACM ICSE*, 2022.
- [14] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, “Fuzzing automatic differentiation in deep-learning libraries,” in *IEEE/ACM ICSE*, 2023.
- [15] Z. Deng, G. Meng, K. Chen, T. Liu, L. Xiang, and C. Chen, “Differential testing of cross deep learning framework apis: Revealing inconsistencies and vulnerabilities,” in *USENIX Security*, 2023.
- [16] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, “Cradle: cross-backend validation to detect and localize bugs in deep learning libraries,” in *IEEE/ACM ICSE*, 2019.

- [17] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audec: Automated testing for deep learning frameworks," in *IEEE/ACM ASE*, 2020.
- [18] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: Testing deep learning libraries via neural architecture fuzzing," in *IEEE/ACM ICSE*, 2022.
- [19] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, 2018.
- [20] X. Zhang, W. Jiang, C. Shen, Q. Li, Q. Wang, C. Lin, and X. Guan, "A survey of deep learning library testing methods," *arXiv:2404.17871*, 2024.
- [21] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *IEEE ISSTA*, 2023.
- [22] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *arXiv:2304.02014*, 2023.
- [23] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *IEEE/ACM ICSE*, 2024.
- [24] S. Fitz and P. Romero, "Neural networks and deep learning: A paradigm shift in information processing, machine learning, and artificial intelligence," in *The Palgrave Handbook of Technological Finance*. Springer International Publishing, 2021.
- [25] D. A. Hashimoto, G. Rosman, D. Rus, and O. R. Meireles, "Artificial intelligence in surgery: promises and perils," *Annals of surgery*, 2018.
- [26] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, 2020.
- [27] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang, "Deepsec: A uniform platform for security analysis of deep learning model," in *IEEE S&P*, 2019.
- [28] X. Ling, L. Wu, J. Zhang, Z. Qu, W. Deng, X. Chen, Y. Qian, C. Wu, S. Ji, T. Luo, J. Wu, and Y. Wu, "Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art," *Computer & Security*, 2023.
- [29] J. Jiang, J. Wu, X. Ling, T. Luo, S. Qu, and Y. Wu, "App-miner: Detecting api misuses via automatically mining api path patterns," in *IEEE S&P*, 2024.
- [30] X. Ling, Z. Wu, B. Wang, W. Deng, J. Wu, S. Ji, T. Luo, and Y. Wu, "A wolf in sheep's clothing: Practical black-box adversarial attacks for evading learning-based windows malware detection in the wild," in *USENIX Security*, 2024.
- [31] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv:1211.5590*, 2012.
- [32] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *MM*, 2014.
- [33] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of machine learning research*, 2018.
- [34] A. Veit and S. Belongie, "Convolutional networks with adaptive inference graphs," in *ECCV*. Springer, 2018.
- [35] A. Gupta, A. Anpalagan, L. Guan, and A. S. Khwaja, "Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues," *Array*, 2021.
- [36] A. Lu, H. Zhang, Y. Zhang, X. Wang, and D. Yang, "Bounding the capabilities of large language models in open text generation with prompt constraints," *arXiv:2302.09185*, 2023.
- [37] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *IEEE/ACM ICSE*, 2024.
- [38] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM TIST*, 2023.
- [39] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv:2308.08155*, 2023.
- [40] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, 2020.
- [41] Y. Chen, J. Wu, X. Ling, C. Li, Z. Rui, T. Luo, and Y. Wu, "When large language models confront repository-level automatic program repair: How well they done?" in *IEEE ICSE 2024 Industry Challenge Track*, 2024.
- [42] H. Abburi, M. Suesserman, N. Pudota, B. Veeramani, E. Bowen, and S. Bhattacharya, "Generative ai text classification using ensemble llm approaches," *arXiv:2309.07755*, 2023.
- [43] M. Zaib, Q. Z. Sheng, and W. Emma Zhang, "A short survey of pre-trained language models for conversational ai-a new age in nlp," in *Australasian computer science week multiconference*, 2020.
- [44] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises & perils," in *IEEE ICSTW*, 2023.
- [45] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, 2020.
- [46] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv:2302.13971*, 2023.
- [47] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, 2022.
- [48] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv:2308.12950*, 2023.
- [49] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "Llm for test script generation and migration: Challenges, capabilities, and opportunities," in *IEEE QRS*, 2023.
- [50] Q. Sun, Z. Chen, F. Xu, K. Cheng, C. Ma, Z. Yin, J. Wang, C. Han, R. Zhu, S. Yuan *et al.*, "A survey of neural code intelligence: Paradigms, advances and beyond," *arXiv:2403.14734*, 2024.
- [51] Y. Liu, G. Deng, Y. Li, K. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, and Y. Liu, "Prompt injection attack against llm-integrated applications," *arXiv:2306.05499*, 2023.
- [52] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "White-box compiler fuzzing empowered by large language models," *arXiv:2310.15991*, 2023.
- [53] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *NDSS*, 2024.
- [54] C. Zhang, M. Bai, Y. Zheng, Y. Li, X. Xie, Y. Li, W. Ma, L. Sun, and Y. Liu, "Understanding large language model based fuzz driver generation," *arXiv:2307.12469*, 2023.
- [55] J. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc, 2006.
- [56] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 international workshop on Mining software repositories*, 2005.
- [57] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, 2023.
- [58] T. Gao, A. Fisch, and D. Chen, "Making pre-trained language models better few-shot learners," *arXiv:2012.15723*, 2020.
- [59] Z. Han, C. Gao, J. Liu, S. Q. Zhang *et al.*, "Parameter-efficient fine-tuning for large models: A comprehensive survey," *arXiv:2403.14608*, 2024.
- [60] M. Jia, L. Tang, B.-C. Chen, C. Cardie, S. Belongie, B. Hariharan, and S.-N. Lim, "Visual prompt tuning," in *ECCV*. Springer, 2022.
- [61] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv:2106.09685*, 2021.
- [62] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and image processing*, 1980.
- [63] HuggingFace, "Hugging face," 2022, <https://huggingface.co>.
- [64] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational api inference," in *ACM ESEC/FSE*, 2022.
- [65] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM computing surveys*, 2020.