LLM-Powered Silent Bug Fuzzing in Deep Learning Libraries via Versatile and Controlled Bug Transfer

KUNPENG ZHANG, Hong Kong University of Science and Technology, China
DONGWEI XIAO*, Hong Kong University of Science and Technology, China
DAOYUAN WU†, Lingnan University, China
SHUAI WANG, Hong Kong University of Science and Technology, China
JIALI ZHAO, Huawei, China
YUANYI LIN, Huawei, China
TONGTONG XU, Huawei, China
SHAOHUA WANG, Central University of Finance and Economics, China

Deep learning (DL) libraries are widely used in critical applications, where even subtle silent bugs can lead to serious consequences. While existing DL fuzzing techniques have made progress in detecting crashes, they inherently struggle to detect silent bugs due to the lack of effective test programs and corresponding oracles.

Building on the observation that historical bug reports contain rich, underutilized information about silent bugs, we leverage large language models (LLMs) to perform *versatile yet controlled* bug transfer for silent bug fuzzing. Specifically, our approach uses LLMs to extract context-aware bug patterns from historical issues, match semantically related Application Programming Interfaces (APIs) using functionality-based embeddings, and synthesize test cases with customized oracles. This enables proactive detection of silent bugs by transferring high-risk contexts and oracle designs from known buggy APIs to functionally similar target APIs. To ensure the reliability of our context-aware bug transfer, we introduce an LLM-powered self-validation module that systematically evaluates the validity of each transferred bug instance. We implement this methodology in a tool named TransFuzz and evaluate it on three mainstream DL libraries: PyTorch, TensorFlow, and MindSpore. TransFuzz successfully discovers 79 previously unknown bugs (12 confirmed as Common Vulnerabilities and Exposures (CVEs)) in 10 bug types, demonstrating its effectiveness and generalizability in migrating DL library bug discovery capabilities.

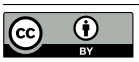
CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**.

Additional Key Words and Phrases: Fuzzing, Deep Learning Library, Large Language Model, Silent Bug

ACM Reference Format:
Kunpeng Zhang, Dongwei Xiao, Daoyuan Wu, Shuai Wang, Jiali Zhao, Yuanyi Lin, Tongtong Xu, and Shaohua Wang. 2026. LLM-Powered Silent Bug Fuzzing in Deep Learning Libraries via Versatile and Controlled Bug

*Corresponding author.
†Work done while the author was associated with HKUST.

Authors' Contact Information: Kunpeng Zhang, Hong Kong University of Science and Technology, Hong Kong, China, kzhangcl@connect.ust.hk; Dongwei Xiao, Hong Kong University of Science and Technology, Hong Kong, China, dxiaoad@cse.ust.hk; Daoyuan Wu, Lingnan University, Hong Kong, China, daoyuanwu@ln.edu.hk; Shuai Wang, Hong Kong University of Science and Technology, Hong Kong, China, shuaiw@cse.ust.hk; Jiali Zhao, Huawei, Shenzhen, China, zhaojiali3@huawei.com; Yuanyi Lin, Huawei, Shenzhen, China, linyuanyi2@huawei.com; Tongtong Xu, Huawei, Shenzhen, China, xutongtong9@huawei.com; Shaohua Wang, Central University of Finance and Economics, Shenzhen, China, davidshwang@ieee.org.



ACM 2475-1421/2026/4-ART150
https://doi.org/10.1145/3798258

Transfer. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 150 (April 2026), 28 pages. https://doi.org/10.1145/3798258

1 Introduction

Deep learning (DL) libraries form the software backbone of many critical applications, such as autonomous driving, medical diagnosis, and finance [Grigorescu et al. 2020; Li et al. 2021, 2023b; Liu et al. 2021; Muhammad et al. 2021; Wang et al. 2023; Wen et al. 2023]. Even subtle bugs in these libraries can lead to incorrect model behavior with high-stakes consequences. While crashes are easily observable and memory errors can be caught by sanitizers, many bugs manifest silently, producing incorrect results without overt signs of failure [Hong et al. 2024; Tambon et al. 2024]. These silent bugs are particularly insidious, as they can lead to erroneous outcomes without being immediately apparent to users or developers. As a result, they are more likely to propagate through downstream applications, causing significant issues that may go undetected.

However, detecting silent bugs is inherently challenging due to the lack of general oracles (i.e., mechanisms used to determine whether a bug has been triggered). Unlike crashes, which exhibit clear and observable symptoms, silent bugs necessitate oracles designed on a case-by-case basis, considering both the specific manifestations of the bug (e.g., incorrect outputs or degraded performance) and the functional characteristics of the involved APIs (e.g., different APIs have different expected outputs) [Hong et al. 2024; Tambon et al. 2024]. This contextual nature makes it challenging, if not impossible, to predefine oracles for all potential silent bugs. Existing fuzzers typically rely on rigid, context-insensitive oracles, such as crashes or CPU-GPU result mismatches, which only capture a small subset of silent bugs. As shown in Table 1, existing tools [Deng et al. 2024; Li et al. 2025b; Zhang et al. 2025b] collectively detect only 3 of 9 representative types of silent bugs (e.g., performance degradation, wrong outputs).

Automated testing for silent bugs is challenging due to the dual requirement of generating test programs and designing effective oracles, as illustrated in Fig. 1. This dual requirement significantly enlarges the testing space, which includes input data, API sequences, and potential oracle behaviors, making exhaustive exploration infeasible. To address this challenge, we should prioritize identifying high-risk APIs and their triggering contexts, focusing testing efforts on these high-stakes scenarios. To that end, an ideal silent bug detector should achieve three key objectives: (1) identifying high-risk APIs, (2) analyzing triggering contexts, and (3) enabling context-aware oracle design.

Our insight stems from the observation that bugs are not isolated incidents; rather, they tend to follow recurring patterns linked to specific functionalities and usage contexts. Through quantitative analysis of historical bug reports, we found that APIs with similar functionalities often exhibit analogous bug behaviors in comparable contexts (see Fig. 3 in Sec. 2.4). Building on this observation, we propose a proactive strategy: generalize high-risk contexts and oracle designs from known buggy APIs to other APIs with similar functionalities. This strategy enables the inference of potentially high-risk APIs and their corresponding triggering contexts from past bug reports, providing actionable guidance for effective oracle design.

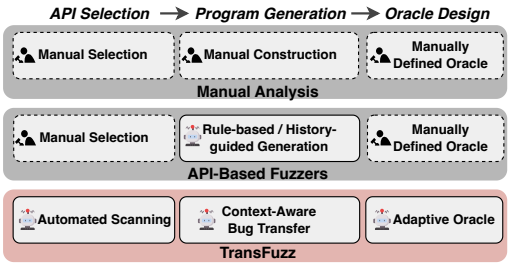


Fig. 1. Testing silent bugs: *Manual Analysis* demands hand-crafted test programs for many APIs; existing *API-Based Fuzzers* struggle with rigid API selection and oracle design.

Based on the above insight, we present TransFuzz, the first automated testing framework for silent bugs in DL libraries. TransFuzz is built on a novel transfer-then-verify pipeline that operates in two stages: bug transfer and bug validation. In the transfer stage, TransFuzz extracts context-aware bug patterns from historical issues, including buggy APIs, triggering contexts, and their oracle designs. These patterns are then transferred to functionally similar target APIs, which are considered potential candidates for similar bugs. Unlike prior work [Deng et al. 2023, 2024; Zhang et al. 2025b], we employ functionality-based API matching, which computes embeddings of API behavior (rather than surface syntax) to identify appropriate transfer targets. Once suitable target APIs are identified, large language models (LLMs) are used to synthesize test programs by adapting the context-aware bug pattern to the new API context, automatically generating custom contexts and oracle designs. While primarily focused on silent bugs, this design can also support regular crash detection.

However, bug transfer alone can introduce false positives, especially when semantic differences between source and target APIs result in behavioral variations. To mitigate this, TransFuzz introduces a rigorous self-validation phase powered by LLMs. In this stage, the LLM is prompted to reflect on the original bug report, the transfer rationale, and the runtime outputs of the test program. It then determines whether the observed behavior constitutes a bug, improving precision for silent bugs.

We implement TransFuzz and evaluate it across three mainstream deep learning frameworks: PyTorch [pyt 2025], TensorFlow [tf 2025], and MindSpore [min 2025]. Although originally designed to detect silent bugs, TransFuzz demonstrates strong scalability and can be seamlessly extended to detect crashes without requiring any modifications. In our evaluation of PyTorch, TransFuzz successfully identifies 31 silent bugs and 25 crashes. Notably, most silent bugs are long-lived and hard to expose: 74% persisted for over three years, and 96% could not be detected by CPU-GPU differential testing, highlighting the necessity of customized oracles beyond simple differential checks. To assess its generalizability, we further transfer the bug patterns mined from PyTorch to TensorFlow and MindSpore, uncovering 23 additional crash bugs. In total, TransFuzz identifies 79 previously unknown bugs in 10 bug types, 12 of which have been assigned CVEs. These results confirm the effectiveness of context-aware bug transfer while maintaining broad applicability. In sum, we make the following contributions:

- We propose a novel *transfer-then-verify* framework that, for the first time, enables automated testing of DL library silent bugs. It systematically extracts context-aware bug patterns and adapts them using functionality-based API matching and LLM-based test synthesis.
- We introduce LLM-powered self-validation that reflects on transfer rationale and test behavior to filter out false positives, enabling reliable detection of silent bugs.
- We implement and evaluate TransFuzz across three DL libraries, discovering 31 silent bugs and 48 crashes (12 CVEs), with high precision on silent bugs and strong cross-framework generalizability.

2 Preliminaries and Related Work

In this section, we introduce silent bugs and their classification, discuss the challenges faced by existing methods in detecting them, and present our insights and pilot experiments.

2.1 Crash Bugs vs. Silent Bugs

Bugs in deep learning (DL) libraries can generally be classified into two categories based on their manifestation: crashes and silent bugs.

Crash bugs are bugs that lead to obvious program failures, such as program crashes, exceptions, infinite hangs, or memory errors. These bugs are typically easier to detect due to their conspicuous symptoms [Christou et al. 2023; Deng et al. 2023, 2022; Wei et al. 2022; Xie et al. 2022; Yang et al. 2023]. For instance, exceptions or crashes often produce error messages or core dumps that can be captured and analyzed, infinite hangs can be identified through timeouts, and memory errors can be detected with sanitizers.

Silent bugs, on the other hand, do not produce immediate or obvious symptoms [Hong et al. 2024; Tambon et al. 2024]. Instead, they lead to incorrect program behavior without crashing or raising errors [Xiao et al. 2022]. These can include incorrect outputs, wrong gradient calculations, or performance degradation. Silent bugs are much harder to detect because they do not manifest through clear failure signals, and are more likely to go unnoticed and persist in downstream applications. Consequently, developing automated techniques to detect silent bugs is a critical yet challenging task.

Classification and Characteristics of Silent Bugs. Building upon existing taxonomies [Tambon et al. 2024], we classify silent bugs into nine categories based on their manifestation, as summarized in Table 1. We first identify eight categories based on distinct observable manifestations, such as discrepancies across execution modes (e.g., *Eager vs. Compiled*), hardware platforms (e.g., *CPU vs. GPU*), or erroneous gradients and outputs (e.g., *Wrong Gradient*, *Wrong Outputs*). These categories reflect clear and recurring failure patterns. We use *Functionality Not Working as Expected* to capture the remaining bugs that do not have the above distinctive signatures or occur infrequently. While our taxonomy does not claim to exhaust all possible forms of silent bugs, it provides a structured and empirically grounded framework for analyzing and detecting these silent bugs in deep learning libraries. Importantly, as will be discussed in Table 1, our analysis further reveals that 8 out of 9 silent bug categories lack a general oracle and thus require context-specific oracle design. Even for the CPU vs GPU category, one of the most common types, general oracles are only effective for detecting numerical discrepancies, but not for other unintended behaviors.

This limitation highlights a fundamental challenge in silent bug detection: silent bugs lack **universal** correctness criteria, as the definition of "correct" outcomes is **context-dependent**, varying with the specific API semantics, input data, and API call contexts. As a result, it is infeasible to define universal oracles that can reliably catch all instances of silent bugs. Instead, effective detection requires designing customized oracles tailored to each use case. For example, to identify *Wrong Outputs*, one must understand the API's semantics and calling contexts, derive the expected output based on the inputs, and validate the actual output using assertions like `torch.allclose()`. Detecting *Wrong Gradients* may involve numerical gradient checks, while identifying *Eager vs. Compiled* discrepancies requires side-by-side execution comparisons under controlled conditions.

2.2 Existing Works and Their Limitations

Based on the characteristics of silent bugs, the ideal automated testing for silent bugs in DL library APIs should have the following key features:

- Be able to identify APIs prone to silent bugs.
- Be able to analyze context information that may trigger these bugs, including API calling context (e.g., specific API call sequences or interactions with other APIs) and input data.
- Be able to design an oracle based on the context information to detect potential bugs.

Despite recent advances, current testing techniques still fall short of meeting these requirements when detecting silent bugs. Specifically, they face the following four challenges:

Challenge I: Inefficient Identification of High-Risk APIs. Current methods struggle to effectively pinpoint APIs susceptible to silent bugs. This inefficiency largely stems from two common

Table 1. Bug detection capabilities by category and tool. Existing tools collectively cover only 3 of 9 silent bug types (~33%), while TransFuzz supports all 9 types.

Category	Bug Type	General Oracle Exists?	FUTURE	DFUZZ	FuzzGPT	TransFuzz
Silent Bug	Eager vs Compiled	Partial[1]	✗	✗	✗	✔
	Eager vs Just-In-Time (JIT)	Partial	✗	✗	✗	✔
	CPU vs GPU	Partial	✔	✔	✔	✔
	Performance degradation	No[2]	✗	✗	✗	✔
	Wrong save/reload	No	✗	✗	✗	✔
	Wrong displayed message	No	✗	✗	✗	✔
	Wrong gradient	Yes	✗	✗	✔	✔
	Wrong outputs	No	✗	✗	✗	✔
	Functionality Not Working as Expected	No	✗	✗	✗	✔
Crash	Crash	Yes	✔	✔	✔	✔

[1] The general oracle can only detect numerical discrepancies, not other unintended behaviors.
[2] Each test case requires a different oracle based on its specific context.

but limited strategies: (1) *Rule-based static matching*. Tools like DFUZZ [Zhang et al. 2025b] and FUTURE [Li et al. 2025b] use manually defined rules to choose target APIs. For example, DFUZZ matches APIs based on parameter type similarity, and FUTURE uses cross-framework mappings. While these methods work in some cases, they rely on manual rules and are ineffective for silent bugs. (2) *Random matching*. For example, FuzzGPT [Deng et al. 2024] randomly selects target APIs, resulting in low testing efficiency and poor coverage of high-risk APIs.

Challenge II: Inefficient Identification of High-Risk Contexts. When testing silent bugs in a target API, both the input space (e.g., data types, values, shapes, attributes) and the interaction space with other APIs must be considered. Each of these spaces is vast, and combining them results in an even larger search space. Methods like random mutation, used by tools such as TitanFuzz and DFUZZ, cannot fully explore this space. Some researchers have used historical bug reports to guide test program generation strategies (e.g., FuzzGPT, FUTURE). However, they typically generate new test cases by embedding the original bug code into prompts. This approach often leads to test programs that only match superficial syntactic patterns of the original bug code, without deeply investigating the root causes of the bug. As a result, the generated tests tend to "overfit" the original bug code. For example, when handling trigger conditions, these methods typically copy the original conditions directly, without adapting them to the characteristics of the target API.

Challenge III: Difficulty in Designing Context-Aware Oracles. Existing DL library fuzzers typically rely on two main oracle strategies, both of which have significant limitations in detecting context-dependent silent bugs. The first observes coarse-grained runtime signals, such as error messages, crashes, or infinite hangs [Deng et al. 2023; Gu et al. 2022; Xie et al. 2022]. The second uses pre-defined oracles for specific types of silent bugs. For instance, some fuzzers include a *CPU vs. GPU* consistency check to detect backend-related discrepancies, while *FuzzGPT* additionally validates gradient correctness.

However, many other types of silent bugs like *Wrong Displayed Message* and *Functionality Not Working as Expected* cannot be captured using such pre-defined oracles. We assess three state-of-the-art fuzzing tools (FuzzGPT, DFUZZ, and FUTURE) in terms of their coverage of nine silent bug categories. As shown in Table 1, even when combining all three methods, only 3 out of 9 scenarios (i.e., ~33%) are covered, leaving 6 scenarios unexplored.

Moreover, manually designing oracles based on the context of each case is impractical, as variations in input data and expected behaviors would require different oracles. For example, with

100,000 test cases, we cannot predict which types of silent bugs may exist in each case. To cover all nine bug types, we would need to design 900,000 unique oracles, which is clearly unfeasible for real-world testing. This gap highlights the need for context-aware oracles that can dynamically detect bugs based on specific API characteristics, input data, and bug manifestations.

Challenge IV: High False Positive Rate. Although some specific types of silent bugs already have expert-driven, general differential testing oracles, such as CPU vs. GPU detection, these methods are not entirely reliable. Output inconsistencies do not necessarily indicate the presence of a bug. Such differences can stem from inherent program behavior, such as the use of random functions in input construction, where outputs naturally vary between runs. Consequently, approaches like TitanFuzz [Deng et al. 2023] and DFUZZ generate a large number of false positives in practice, requiring significant manual effort to analyze and verify each case.

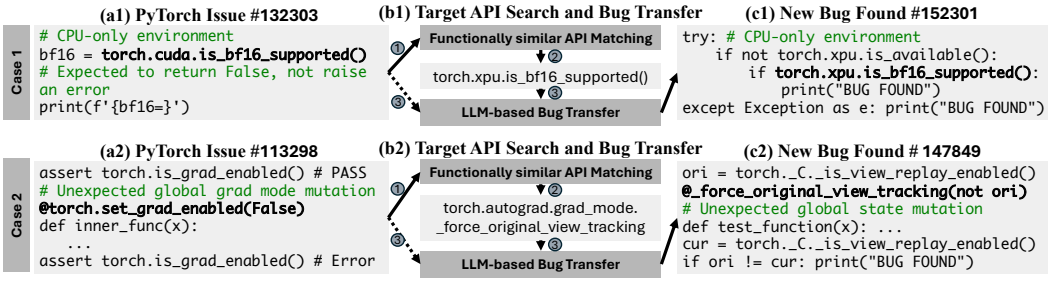


Fig. 2. Two Examples of Silent Bug Transfers.

2.3 Motivating Insights and Examples

The limitations outlined above arise from a core shortcoming: existing methods treat bug discovery as isolated, API-agnostic tasks, without systematically leveraging historical knowledge or cross-API semantic patterns to guide testing and oracle design. Our key innovation lies not merely in applying LLMs to fuzzing workflows, but in designing a *pattern-transfer-driven* framework that bridges known bugs with potentially vulnerable APIs through structured reasoning. Our approach is grounded in two key insights: *Insight 1*: Historical bug reports contain a wealth of unexploited information about silent bugs. *Insight 2*: API bugs, especially in API implementations, are rarely isolated. When APIs share similar functionality, they often follow comparable design and implementation patterns, leading to analogous vulnerabilities. For instance, numerical computation errors may recur across functionally similar APIs when handling analogous boundary conditions. In Sec. 2.4, we further conduct a quantitative analysis to validate this observation.

Based on Insight 1, we can extract known bug APIs, their triggering contexts, and oracle designs from historical bug reports. Drawing from Insight 2, we can identify candidate target APIs that are functionally similar to the known buggy API. These candidates may suffer from analogous silent bugs (*Challenge I*). We then instruct the LLM to generate custom triggering contexts and oracle designs based on the characteristics of the target API, integrate them into a test program, and execute the program to verify whether similar bugs exist (*Challenges II & III*). Finally, to reduce false positives (*Challenge IV*), we employ the LLM to reflect on the entire transfer process, evaluating whether the inferred bug pattern is logically and semantically plausible in the new context. By transferring known bug patterns across similar APIs with LLM-powered adaptation and validation, we form a closed-loop pipeline that systematically uncovers silent bugs.

Motivating Example. As shown in Fig. 2, we illustrate our approach using two representative *original* silent bug issues, which serve as source bug patterns for transfer. Case 1 corresponds to

an environment-related logic bug in `issue #132303`, where `torch.cuda.is_bf16_supported` raises an exception instead of returning `False` when executed in a CPU-only environment. Case 2 corresponds to a global state mutation bug reported in `issue #113298`, in which certain context-management APIs unexpectedly modify global execution states (e.g., gradient mode), leading to silent semantic inconsistencies.

- **Addressing Challenge I (High-Risk API Identification):** Instead of random testing, we transfer bug patterns extracted from original issues to other functionally similar APIs to identify high-risk testing targets. For Case 1, we match `torch.xpu.is_bf16_supported` as a target API because it performs analogous hardware capability checks under different device backends. For Case 2, we identify `_force_original_view_tracking` as a target, as it serves a similar role in managing global execution states.
- **Addressing Challenges II & III (Context & Oracle Design):** After identifying target APIs for bug pattern transfer, we further transfer and adapt the associated triggering contexts and oracle designs from the original issues. Existing fuzzers such as TitanFuzz and FreeFuzz rely on general-purpose oracles (e.g., crashes or CPU-GPU output consistency), which cannot detect these logic bugs, as they exhibit fully consistent behavior across CPU and GPU executions. Identifying such logic bugs typically requires developers to manually design tailored triggering contexts and oracles for each case, which is labor-intensive and difficult to scale. In Case 1, our tool extracts specific triggering contexts (e.g., "CPU-only environment") and custom oracles (e.g., "Check if return is `True`" or "Catch specific warning messages") from historical issue and related patches and adapts them to the target API. In Case 2, it captures the implicit assumption that context managers should not mutate global states, and designs an oracle that checks whether execution states (i.e., view-replay mode) remain unchanged before and after API invocation. Consequently, an oracle trigger serves as a diagnostic signal for a potential bug.
- **Addressing Challenge IV (High False Positives):** To ensure the correctness of transferred patterns, we apply an LLM-powered self-validation step. For example, in Case 2, our tool reasons about whether a detected global state change reflects a true semantic violation or an intended design choice. Only behaviors that are inconsistent with documented semantics are reported as bugs. Detailed mechanisms and additional examples are presented in Sec. 3.5.

2.4 Pilot Experiments

To investigate our hypothesis that bugs in DL libraries exhibit recurring patterns tied to specific API functionalities and usage contexts, we conduct a pilot study analyzing real-world bug reports from PyTorch. Our goal is to assess whether similarities in API functionality and triggering context correlate with similarities in how bugs manifest, particularly in terms of oracle design.

Data Collection. We collect all GitHub issues labeled as "correctness (silent)" from the official PyTorch repository, which indicate silent correctness errors. After removing irrelevant discussions (e.g., feature requests), we randomly sample 100 bug reports for in-depth analysis.

Information Extraction. For each selected issue, we use GPT-4.1-mini to extract three key attributes: (1) *bug API functional description*, (2) *triggering context*, and (3) *oracle design*. These attributes are then manually verified, and embedded into vectors using the `text-embedding-3-small` model. As a result, each of the 100 issues is represented by a triplet of embedding vectors: (API functional description, usage scenario, oracle design).

Similarity Analysis and Visualization. We compute the cosine similarity between every pair of issues across all three dimensions, yielding 9,900 pairwise comparisons. Each comparison includes three similarity scores: API functional, triggering context, and oracle design similarity. To visualize

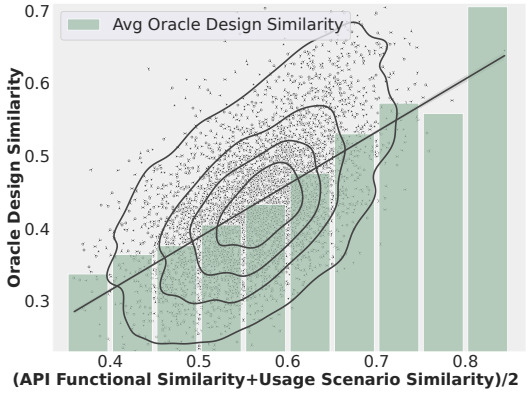


Fig. 3. Oracle Design Similarity vs. Functional and Usage Scenario Similarity.

these relationships, we map each pair into a 2D space: the x-axis is the average of API functional and usage scenario similarities *(x = 0.5 × API functional similarity + 0.5 × usage scenario similarity)*, and the y-axis is the *oracle design similarity*. Fig. 3 shows the resulting scatter plot, along with regression lines, density contours, and bar charts that display the mean oracle design similarity within 0.05-wide x-axis bins.

Observation. The plots reveal a clear trend: *when two issues involve APIs with similar functionalities and triggering contexts, they are more likely to share similar bug manifestations and oracle designs.* We further quantified this using Pearson correlation analysis, yielding a Pearson correlation coefficient of $r = 0.561$ ($r > 0$ signifies a positive association) and a p-value of 0.000 ($p < 0.05$ denotes statistical significance). This result confirms a statistically significant positive correlation between API functionality/context and bug manifestation.

3 Design

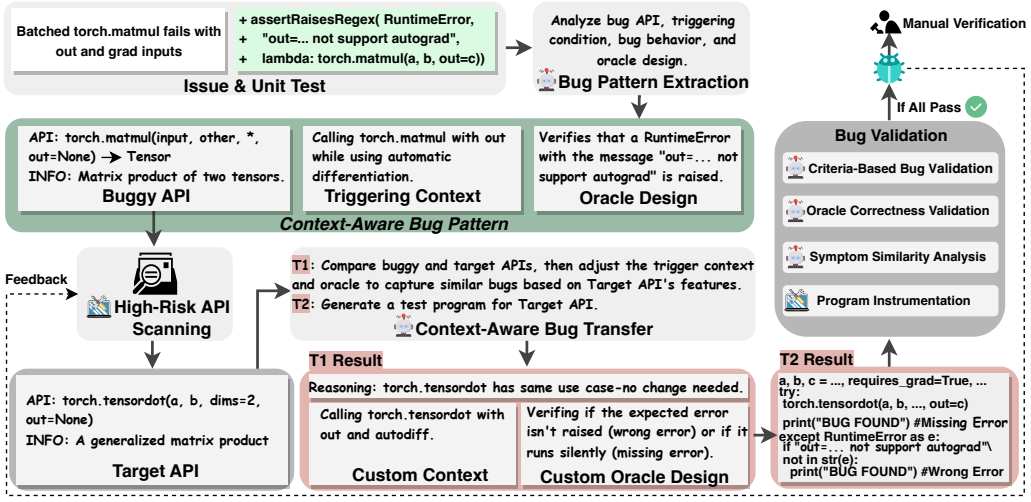


Fig. 4. The Workflow of TransFuzz, which consists of four main stages: Bug Pattern Extraction, High-Risk API Scanning, Context-Aware Bug Transfer, and Bug Validation.

3.1 Overview

To address the challenges highlighted in Sec. 2, we present TransFuzz, a novel transfer-then-verify framework that fuzzes silent bugs through *versatile yet controlled* bug transfer. Fig. 4 shows the workflow of TransFuzz. It consists of four key components:
① *Context-Aware Bug Pattern Extraction*: TransFuzz begins by mining historical bug reports. For each bug, it extracts a *context-aware bug pattern* that encapsulates three critical elements: the buggy API, the triggering context, and the oracle design. These patterns serve as structured templates to guide subsequent bug transfer.
② *Functionality-Based API Matching*: To generalize bug patterns beyond the original API, TransFuzz identifies functionally similar APIs by computing semantic similarity between context-free functional descriptions of the original API and other library APIs. This produces a prioritized queue of candidate APIs (i.e., the similar API queue) likely to exhibit similar vulnerabilities.
③ *Bug Transfer-Driven Fuzzing*: Given a bug pattern and the similar API queue, TransFuzz generates targeted test programs. It selects target APIs using a feedback-driven strategy based on the similar API queue, enabling efficient identification of APIs likely to exhibit similar bugs. For each target API, TransFuzz uses LLMs to adapt the bug pattern by rewriting the triggering context and redesigning the oracle to match the new API's semantics. This yields customized, executable test programs.
④ *LLM-Powered Self-Validation*: Since LLM-generated oracles are not always reliable, TransFuzz performs post-execution validation using LLMs to minimize false positives. By reflecting on the original issue, transfer rationale, and runtime behavior of the generated tests, TransFuzz significantly improves the reliability and precision of bug detection.

3.2 Context-Aware Bug Pattern Extraction

Context-Aware Bug Pattern. Extracting known bug information from historical bug reports forms the foundation of effective bug transfer. The quality of this extraction directly impacts the success of the transfer process. We introduce the concept of *context-aware bug pattern*, which encapsulates all critical elements involved in triggering and detecting silent bugs, including (i) *Bug API*: The specific API where the bug was originally discovered; (ii) *Triggering Context*: The conditions or actions that activate the bug, such as invalid inputs or a particular sequence of API calls; and (iii) *Oracle Design*: An automated mechanism for detecting the bug in code, enabling identification without manual inspection.
Why is it necessary to extract context-aware bug patterns? A straightforward approach to bug transfer is to directly use historical bug code to guide test program generation, as seen in tools like FuzzGPT and FUTURE. However, this method may overlook that bug reports often contain irrelevant details and contextual noise, such as unrelated variables or redundant control structures. If we prompt LLMs to generate test programs directly from such noisy reports, aiming at triggering similar bugs in target APIs, the models may struggle to abstract the core bug logic. Instead, they tend to overfit on superficial syntactic elements, failing to capture and transfer the essential characteristics of the bug. As a result, the generated test programs could be often ineffective.

To address this, we propose context-aware bug patterns as an intermediate representation in the bug transfer pipeline. Instead of directly transferring from a bug report to a test case, we adopt a staged approach: *Original Bug Report* → *Context-Aware Bug Pattern* → *New Bug in the Target API*. This enhances the accuracy and generalizability of bug transfer. By first distilling a precise bug pattern, we enable the LLM to generate a customized triggering context and oracle that align with the unique characteristics of the target API, ultimately improving the likelihood of uncovering potential similar bugs.

Our Method. In this paper, we use PyTorch as an example to demonstrate our approach.[1] To enable effective bug transfer, we first collect real-world bug instances that are both reproducible and user-relevant. Specifically, we crawl issues and pull requests (PRs) from the PyTorch repository, focusing on those that reflect bugs observable from the user's perspective. Given that PyTorch has over 50,000 issues, analyzing each one would consume an excessive number of LLM tokens. We thus focus only on issues that have corresponding PRs, ensuring that the issue is a confirmed bug that has been addressed and fixed. In the future, we could leverage LLMs to analyze the issues further, such as identifying whether the bugs are user-triggerable, and thereby extract additional bug patterns.

Building on this curated dataset, we then leverage the advanced code comprehension capabilities of LLMs [Brown et al. 2020; Devlin et al. 2019; Feng et al. 2020; Li et al. 2023a; Yadavally et al. 2023] to extract context-aware bug patterns from both the issue discussions and the associated code changes, using a carefully designed prompt to guide systematic analysis. As illustrated in Fig. 13 (in our extended version [Zhang et al. 2026]), this prompt guides the LLM to analyze issue reports and code changes, identify the exact API involved, clarify the conditions required to trigger the bug, contrast the expected versus actual behaviors, and pinpoint the oracle used to detect the bug. Moreover, the prompt further directs the model to generate a user-facing Python test program that can reproduce the bug, ensuring that the extracted patterns are actionable and grounded in real-world usage scenarios.

3.3 Functionality-Based API Matching

After extracting the context-aware bug pattern from the original issue, our next step is to identify candidate transfer targets, which are other APIs that might exhibit similar bugs. Our analysis of historical bug reports reveals a key empirical pattern: APIs with similar functionalities tend to exhibit analogous bugs under comparable contexts; see Fig. 3 in Sec. 2.4. Leveraging this insight, we use functional similarity as a guiding principle to identify candidate transfer targets. Unlike DFUZZ, which performs transfer mainly at the *input level* based on syntactic properties (e.g., input types and signatures), TransFuzz performs transfer at the *API (functional) level* by modeling semantic functionality, enabling the transfer of bug-triggering logic and oracle designs even across APIs with different signatures. This broader transfer capability is reflected in our evaluation: Table 1 shows that DFUZZ covers only 1 of 9 silent bug categories, whereas TransFuzz supports all 9, and Table 5 further indicates that DFUZZ detects only one silent bug, compared to 31 detected by TransFuzz.

To implement this matching process, we start by automatically scanning the target library to collect all available APIs, including their function names and module paths. We then crawl documentation and parameter details for each API. Next, leveraging well-crafted prompts (see Fig. 14 in our extended version.), we employ LLMs to distill these documents, generating context-free functional descriptions that capture input parameter types and core operation workflows. Once we have these descriptions, we encode them into fixed-dimension vectors using embedding models like the OpenAI embedding API, such as *text-embedding-3-small*. All embeddings are stored systematically within an API embedding database for easy retrieval. Finally, using the API embedding database, we compute cosine similarity scores for each context-aware bug pattern, identifying the top 1,000 APIs most similar to the original bug API. This forms the *similar API queue*, which is used for subsequent test case generation.

[1]We also transfer bug patterns from PyTorch to the other two DL libraries for crash detection; see Sec. 4.2.

3.4 Bug Transfer-Driven Fuzzing

Having laid the groundwork in previous sections, we now present our core methodology of bug transfer-driven fuzzing. Specifically, in Sec. 3.2, we extract context-aware bug patterns for each issue, including details such as the original bug API. In Sec. 3.3, we generate a similar API queue for each original bug API based on context-free functional similarity. We integrate these components into a fuzzing framework that performs bug transfer-guided test generation.

Overview. Our bug transfer-driven fuzzing consists of two components: *Feedback-Driven API Selection* and *Context-Aware Bug Transfer*. Leveraging the similar API queue, we propose a feedback-driven API selection strategy to efficiently identify target APIs likely to exhibit similar bugs. For each target API, we design context-aware triggering context and oracles based on API functionality to detect analogous bugs and automatically generate test programs for validation.

Alg. 1 outlines the entire fuzzing process. Given an issue and its fix, our method first extracts a context-aware bug pattern via LLMs (Sec. 3.2) and builds a queue of similar APIs (Sec. 3.3) in lines 2-3. The fuzzing loop iteratively selects up to 10 untested APIs from the queue (line 6), and for each newly discovered buggy API, adds its top-10 similar APIs to expand the search (lines 7-9). For each target API, the system generates a test by transferring the bug pattern and runs it (lines 10-12). If a bug is found and verified, the API is recorded (lines 13-15). The process continues until no new bugs are detected, enabling iterative, feedback-driven bug discovery.

Algorithm 1: Bug Transfer-Driven Fuzzing.

```
   Input: An issue and its corresponding PR
1  init := true
2  bug_pattern ← LLM_extract(issue, PR)
3  api_queue ← API_match(bug_pattern.api)
4  found_new_bug_api ← ∅
5  while init or found_new_bug_api ≠ ∅ do
6      targets ← top-10 untested APIs from api_queue
7      foreach a ∈ found_new_bug_api do
8          add top-10 similar APIs of a to targets
9          remove a from found_new_bug_api
10     foreach a ∈ targets do
11         p ← Transfer(bug_pattern, a)
12         run(p)
13         if BUG FOUND then
14             if self_validation(issue, p) then
15                 add a to found_new_bug_api
16     init := false
```

Feedback-Driven API Selection. Under limited testing resources, we propose a feedback-driven API selection strategy to efficiently identify APIs likely to harbor similar bugs based on the similar API queue. We iteratively conduct batch testing following the similar API queue, as shown in Fig. 4. In each round, we select the top N untested APIs (where N is the window size, set to 10 based on the results in Sec. 4.5) with the highest similarity scores for bug transfer. If any bugs are detected in a round, the next batch proceeds; otherwise, the process terminates early. This strategy allows us to quickly focus on high-risk areas while avoiding wasted resources on low-yield candidates.

When a transferred test program for a target API uncovers a new bug, we classify the functionality of that API as high-risk. We then use the embedding of this target API as an anchor to identify and

test the 10 most similar APIs. This feedback loop enables us to go beyond the static ranking of the initial queue and adaptively explore APIs that are more likely to contain similar bugs, guided by actual fuzzing outcomes.

Context-Aware Bug Transfer. To enable effective migration of the original bug to target APIs, we introduce a context-aware bug transfer approach. Fig. 16 (in our extended version) illustrates the prompt that completes this entire synthesis process in a Chain-of-Thought manner. Specifically:

First, we analyze the functional semantics and usage scenarios of both the original bug API and the target API. This comparison highlights their differences, providing a robust semantic foundation for subsequent bug transfer reasoning. Next, assuming that the target API triggers similar bugs, we prompt the LLM to infer its potential triggering context and abnormal behavior, such as returning error codes, throwing exceptions, or causing resource leaks. Importantly, even for similar bugs, the precise triggering contexts often differ across APIs. For instance, an edge-case bug in API A may be triggered when *input = 0*, while the equivalent edge case in API B may instead occur at *input = -1*.

Then, based on the expected abnormal behavior, TransFuzz generates a corresponding oracle to catch it. Oracle design is guided by both API characteristics and bug types, requiring careful consideration of the specific functional properties of the API and the nature of the bug. For example, oracle designs for "wrong displayed message" and "wrong gradient" are fundamentally different. Typically, the oracles include assertions on return values, checks for exception types and error messages, and resource state confirmations, thereby ensuring clear test verdicts and strong coverage.

Finally, TransFuzz synthesizes complete test programs by integrating the triggering context, API invocation sequence, and oracle design. These programs are executable out-of-the-box and serve to validate whether the hypothesized bug is indeed reproducible in the target API.

3.5 LLM-Powered Self-Validation

After completing the bug transfer process, we generate the corresponding test programs tailored to the target APIs. When executed, if a test program outputs *"BUG FOUND"*, indicating that the oracle has been triggered, it suggests the presence of a potential bug. However, due to the inherent randomness in LLM-generated code, the oracles themselves may contain inaccuracies or over-generalizations, leading to false positives. As a result, not all triggered oracles reflect real bugs in the target API. Therefore, further validation is required to confirm whether they are false positives.

Insight. In theory, if comprehensive and precise documentation clearly defined the expected behavior of an API under all conditions, LLMs would be able to identify false positives. However, the real-world use of APIs is highly complex, often involving numerous edge cases. In practice, documentation typically provides only a brief overview of API functionality and descriptions of input and output parameters. As a result, directly relying on LLMs to determine whether a case is a false positive often leads to a high rate of misclassification. As shown in Table 11 (in our extended version), we also explore the direct use of LLMs in the curated dataset to determine whether a transferred test program triggers a real bug. Experimental results show that the precision of this approach is only 30.77%. It is important to note that 28.20% of the samples in this dataset are real bugs. This means that even a naive strategy that predicts every sample as a real bug would achieve a precision of 28.20%.

To reduce false positives, it is crucial to establish reliable criteria for judgment. We observe that fixed bugs serve as ideal reference points, since they have been identified by developers. By extracting the **bug determination criteria** (i.e., the core rationale behind labeling specific program behaviors as bugs) from these resolved cases, we can guide automated reasoning. More importantly, by establishing these criteria, we transform the subjective task of determining whether a test triggers a real bug into an objective classification with clear decision grounds. This novel approach mitigates LLM hallucinations and forms the foundation of our validation process. Leveraging

the bug determination criteria allows us to analyze the runtime behavior of transferred tests to determine whether they are false positives.

Overview. Based on the above insight, we propose an *LLM-powered self-validation* approach that automatically filters false positives without any human involvement. Fig. 5 illustrates the overall workflow, which consists of two tightly coupled components: ***(1) Instrumentation*** and ***(2) Bug Transfer Validation***.

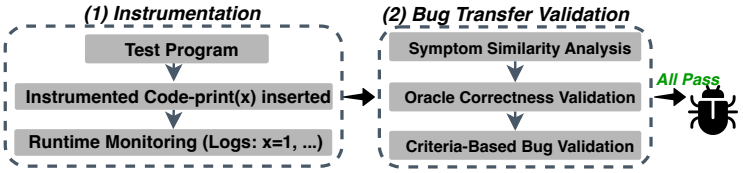


Fig. 5. Overview of LLM-powered self-validation.

Given a transferred test program that triggers an oracle (i.e., outputs *"BUG FOUND"*), directly trusting this result is unreliable. In practice, an oracle may be triggered by unintended program behaviors rather than the intended manifestation of a real bug, due to corner cases, environment-specific effects, or deficiencies in the oracle design. To disambiguate these scenarios, we therefore examine the program's *actual runtime behavior*. To this end, we introduce automated *Instrumentation* to capture fine-grained runtime states during execution. Instrumentation produces execution logs that record the program's concrete runtime behavior, such as relevant variable states. These logs serve as the factual input to the subsequent *Bug Transfer Validation*, grounding the LLM's reasoning in concrete execution evidence.

Our *Bug Transfer Validation* consists of three complementary checks to determine whether the transferred program triggers a real bug: *(i) Symptom Similarity Analysis*: Given the functional similarity between the original and target APIs, comparable behavior is expected when similar bugs are triggered. Thus, we assess the behavior similarity between the original issue and the transferred program. *(ii) Oracle Correctness Validation*: Assuming the target API is bug-free, we verify that the oracle is logically valid and not triggered by environmental noise or LLM-induced hallucinations. *(iii) Criteria-Based Bug Validation*: Extracting the bug determination criteria from the original issue and using them to assess whether the transferred program is a false positive. Only when all checks are satisfied do we conclude that the transferred program triggers a real bug.

1) Instrumentation. The role of instrumentation is to bridge the gap between oracle-level signals and semantic bug reasoning. While an oracle only reports that a bug condition is met, instrumentation reveals how the program actually reaches that condition at runtime. This includes the executed path, the specific oracle that is triggered, and the runtime states that lead to the trigger. Such execution evidence enables the LLM to determine whether an oracle trigger reflects the intended manifestation of a real bug or results from unintended side effects. Therefore, instrumentation is applied uniformly across all subsequent validation stages. Specifically, the execution logs generated by the instrumentation are leveraged in the prompts for Symptom Similarity Analysis, Oracle Correctness Validation, and Criteria-Based Bug Validation.

We implement instrumentation via automated Abstract Syntax Tree (AST) transformation on the transferred test program. The transformation is lightweight, fully automated, and preserves original program semantics. It supports the following language constructs:

Variable Assignments: We cover ordinary assignments (*a = 1*), attribute assignments (*obj.attr = 1*), index assignments (*lst[0] = 2*), and multi-variable unpacking (*a, b = 1, 2*). For each variable's first assignment, we insert a print statement (e.g., *print("var =", var)*).

Exception Handling: For exception variables captured via *except ... as e*, a print statement is injected at the entry of the exception handler to log the exception state.

Context Managers: For context manager variables in *with ... as w* statements, we place instrumentation at the beginning of the controlled block to monitor the runtime value of the acquired resource.

Conditional Expressions: We implement recursive parsing for complex nested structures within *if* conditions, instrumenting only those sub-expressions that can be independently evaluated. For instance, in the condition *if torch.isnan(x.grad).any()*, instrumentation is applied to *x.grad*, *torch.isnan(x.grad)*, and the complete condition itself, ensuring that only expressions with actual runtime values are captured. Additionally, our method supports extraction across multi-level call chains. For expressions such as *a.b().c[0].d()*, instrumentation is applied stepwise to *a.b()*, *a.b().c*, *a.b().c[0]*, and finally *a.b().c[0].d()*.

Table 2. Key IR identifiers and their meanings.

IR Identifier	Description
`VarDef(<type>)`	Defines a variable of given type, e.g., `tensor`.
`APICall(<api>)`	Represents a function call and its return value.
`OracleCheck(<type>)`	Specifies assertions, e.g., *ValueCorrectness*.
`condition=`	Specifies comparison logic, e.g., `Compare(v1,v2)`.

2) Symptom Similarity Analysis. Since our fuzzing framework is based on migrating bug patterns across functionally similar APIs, it is reasonable to expect a high degree of behavioral consistency between the original issue and its transferred programs. Accordingly, our validation focuses on whether the transferred program preserves the essential bug pattern of the original issue under a different API context. To this end, we organize the analysis as a two-step validation pipeline that progressively filters incorrect transfers while avoiding unnecessary fine-grained analysis. We first check whether the original issue and the transferred case belong to the same bug type, which serves as a coarse-grained prerequisite for further validation. Based on the identified bug type, we then apply different checks: for mismatch bugs, we verify whether the transferred program triggers a real mismatch; for all other bug types, we examine whether the two cases exhibit the same bug pattern by comparing their trigger conditions, runtime behaviors, and oracles.

We begin by determining whether the original issue and the transferred test program belong to the same bug type. A discrepancy at this stage signals an incorrect bug transfer and the case is immediately filtered. Directly prompting LLMs to compare bug types often leads to inconsistent or self-justifying outputs. To make this process objective and reproducible, we introduce an intermediate representation (IR) that formally defines seven common bug types. Key IR identifiers are summarized in Table 2. This IR-based formulation transforms subjective semantic comparison into structured pattern matching, effectively reducing LLM hallucination. The prompt used for this classification is shown in Fig. 20 (in our extended version). As an example, the IR definition of a device inconsistency bug is as follows:

Device_Inconsistency ::=
  `VarDef(tensor)[device=cpu:*]` $\rightarrow v_{cpu} \wedge$
  `VarDef(tensor)[device=cuda:*]` $\rightarrow v_{gpu} \wedge$
  `OracleCheck(ValueCorrectness)(`
    `condition=Compare(`v_{cpu}, v_{gpu}`)`
) $\rightarrow$ `FAIL`

Other bug types are defined in a similar manner in our extended version.

For mismatch bugs, the oracle is typically simple and reliable, as it only checks for output inconsistencies of the same program across different environments. As a result, we do not perform additional fine-grained bug pattern analysis, but only need to double-check whether this is a mismatch bug based on the prompt in Fig. 21 (in our extended version). Otherwise, for all non-mismatch bug types, oracle reliability is often lower and a stricter semantic comparison is required. In such cases, we assess whether the original issue and the transferred test program share the same bug pattern from three complementary perspectives: bug trigger conditions, runtime behaviors, and oracle designs. Leveraging LLMs together with instrumented execution logs, we analyze the consistency of these dimensions between the source and transferred cases. The prompt used for this analysis is shown in Fig. 22 (in our extended version).

Fig. 6 illustrates how symptom similarity analysis filters out a false positive caused by an incorrectly transferred oracle. First, at the code level, the original *torch.clamp* issue is classified as a `Functional_Defect`, as its oracle checks whether an empty tensor is returned when both *min* and *max* are *None*. The transferred program for *torch.clip* similarly validates functional correctness by checking the runtime outcome, either by inspecting whether the output is empty or whether the raised error message contains a target string. Since both oracles reason about functional behavior, the transferred program passes the coarse-grained bug-type consistency check.

TransFuzz then conducts a finer-grained analysis using instrumented execution logs to assess whether the original and transferred programs exhibit the same bug pattern. The analysis uncovers a clear discrepancy: the original oracle characterizes the bug by detecting an empty output tensor, whereas the oracle actually triggered in the transferred program checks whether the raised error message matches a specific string. As these two oracles encode fundamentally different bug patterns, TransFuzz identifies the transferred case as a false positive and filters it out. Further inspection reveals that the error-message-based oracle in the transferred program is incorrect, as it is introduced by the LLM mistakenly treating the patched, correct behavior of *torch.clip* (raising a *RuntimeError*) as a bug oracle.

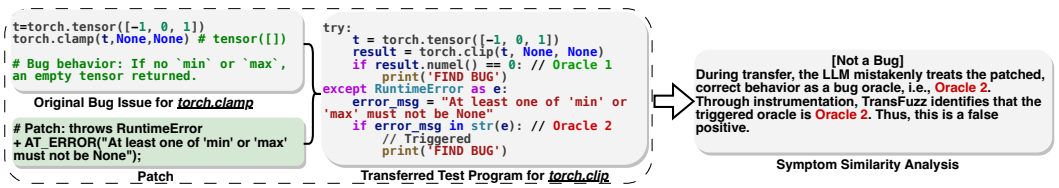


Fig. 6. False positive filtered by symptom similarity analysis. The transferred program for `torch.clip` incorrectly checks error messages instead of empty output, revealing a semantic mismatch from the original `torch.clamp` bug.

3) Oracle Correctness Validation. For the cases that pass the checks in symptom similarity analysis, we use a reverse hypothesis validation method to analyze the correctness of the oracle design, specifically to determine whether it might be triggered incorrectly. We begin by assuming that the target API does not contain a bug similar to the one in the original issue. Under this assumption, if the oracle in the transferred test case still gets triggered, this implies a flaw in the oracle's design, as it should not flag a problem in the absence of a bug. The prompt for this analysis is shown in Fig. 23 (in our extended version). As shown in Fig. 7, we use an example to demonstrate how oracle correctness validation identifies false positives induced by invalid semantic assumptions in the transferred oracle. The original bug originated from *torch.mul* violating the commutative property under the *torch.compile* mode, where the oracle specifically detects whether

$f(a, b) \neq f(b, a)$. When this bug pattern is transferred to *torch.fmod*, the test similarly flags non-commutative behavior as a bug. However, *fmod(x, y)* and *fmod(y, x)* are mathematically non-commutative by definition, meaning the oracle can be triggered even when the implementation is fully correct. Oracle correctness validation identifies that the oracle encodes an invalid semantic assumption for the transferred API and therefore filters this case as a false positive.

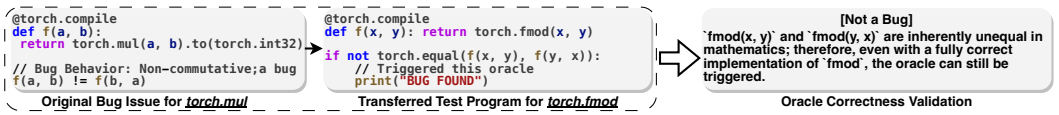


Fig. 7. False positive filtered by oracle correctness validation. The non-commutative property holds for `torch.mul` but not for `torch.fmod` by mathematical definition.

4) Criteria-Based Bug Validation. For the cases that pass the checks in oracle correctness validation, we use a criteria-based bug validation method to ultimately determine whether this is a real bug. The core of this analysis lies in extracting bug determination criteria from the original issue, then using these criteria to assess whether the transferred test program genuinely triggers a bug in the target API. To achieve this, the process unfolds in three steps.

First, assess whether reliable bug determination criteria can be extracted from the original issue by evaluating four dimensions: API-level bug relevance, presence of a demo, negative feedback, and bug complexity, with prompts shown in Fig. 17, Fig. 15, Fig. 18, and Fig. 19 in our extended version, respectively. An issue is considered suitable for extraction only if it satisfies all four checks.

Next, analyze the original issue and, based on the functional characteristics of the API, justify why the observed behavior should be classified as an API bug rather than expected behavior. This involves identifying the specific functional requirement or intended behavior that is violated, forming the foundational bug determination criteria. The prompt guiding this analysis is shown in Fig. 24 (in our extended version).

Finally, based on the execution outputs, analyze the transferred test program to determine why the oracle is triggered. Specifically, why *"BUG FOUND"* is printed. Then, using the bug determination criteria extracted from the original issue as a reasoning framework, evaluate whether the observed behavior in the target API is a real bug. This involves assessing whether the oracle's trigger reflects a real violation of expected functionality, rather than an inconsistency in test design or environmental factors. The prompt guiding this analysis is provided in Fig. 25 (in our extended version). To enhance accuracy, we introduce a debate mechanism. After receiving the initial prompt response, the LLM will adopt a counterpoint perspective, questioning the original interpretation (i.e., "Please challenge this from the opposing viewpoint"). Ultimately, the final assessment will summarize whether this constitutes a false positive, based on both perspectives (i.e., "Based on both viewpoints, summarize whether this is a false positive"). As shown in Fig. 8, we present an illustrative example of how criteria-based bug validation identifies and filters false positives by verifying the applicability of the original bug's logic to the transferred API. From the original *torch.argmax* issue, we extract a bug criterion stating that since *argmax* is non-differentiable, triggering an *INTERNAL ASSERT FAILED* error or observing *requires_grad = True* signals a bug. This criterion is then applied in a transferred test for torch.amax, with the oracle checking whether the output tensor propagates gradients. However, unlike *argmax*, *amax* is differentiable by design, and *requires_grad = True* is therefore expected behavior rather than a bug symptom. Criteria-based bug validation detects that the transferred case violates the semantic preconditions underlying the original criterion and correctly filters it as a false positive.

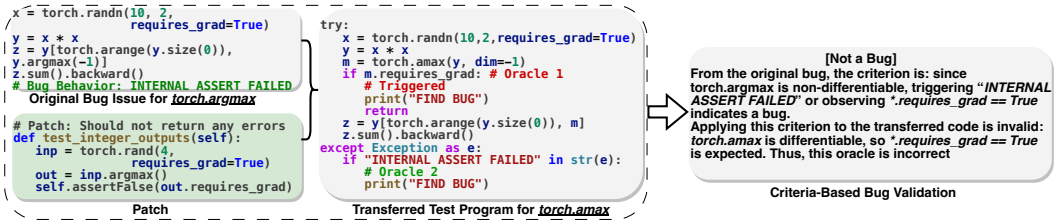


Fig. 8. False positive filtered by criteria-based bug validation. `torch.argmax` is non-differentiable by design, but `torch.amax` is differentiable.

4 Implementation and Evaluation

The core testing process of TransFuzz is implemented in Python, consisting of about 3,600 lines of code. Our implementation provides automated issue analysis, API analysis, and test generation to support our evaluation. To evaluate the effectiveness of TransFuzz, we focus on the following research questions (RQs):

- **RQ1**: How many previously unknown bugs and distinct types of bugs can TransFuzz discover?
- **RQ2**: How does TransFuzz compare with existing tools in terms of effectiveness?
- **RQ3**: How many detected bugs are real?
- **RQ4**: How are the hyperparameters determined?
- **RQ5**: What are the results of the manual analysis of prompt effectiveness?
- **RQ6**: What is the cost of using TransFuzz?

4.1 Experimental Setup

Environment. All experiments are performed on a machine equipped with 256 GB of RAM, an NVIDIA RTX A6000 GPU, and Ubuntu 20.04 as the operating system.
LLMs. We use o3-mini [o3m 2025] for *Context-Aware Bug Pattern Extraction*. Given that generating context-free functional descriptions in *Functionality-Based API Matching* is relatively straightforward, we choose to use GPT-4o mini [gpt 2024]. Since *Bug Transfer-Guided Test Generation* requires generating a large number of test cases, we adopt GPT-4o mini for efficiency. For *LLM-Powered Self-Validation*, we utilize GPT-4.1 mini [gpt 2025] for its stronger reasoning capabilities.

4.2 RQ1: Bug Discovery

Although TransFuzz is originally designed to detect silent bugs, it exhibits strong scalability and can be seamlessly extended to detect crashes without requiring any modifications. We evaluate its effectiveness in both scenarios across three mainstream deep learning frameworks: PyTorch [pyt 2025], TensorFlow [tf 2025], and MindSpore [min 2025].
Silent Bug Detection in PyTorch. We conducted silent bug detection on PyTorch versions 2.6.0 and 2.7.0, utilizing 7,466 historical issues as source data. Table 3 summarizes the results of silent bug detection in PyTorch using TransFuzz. A total of 31 silent bugs were identified across nine categories, with the majority belonging to three primary types: Functionality Not Working as Expected (14 bugs), Wrong Displayed Message (7 bugs), and Wrong Outputs (4 bugs). Each bug was identified by transferring patterns from a known source issue and subsequently reported as a new issue. Notably, some source issues triggered multiple bugs, indicating the presence of reusable bug patterns. Each of the remaining categories (i.e., CPU vs GPU, Eager vs Compiled, Eager vs JIT, Performance Degradation, Wrong Save/Reload, Wrong Gradient, and Wrong Outputs) accounted

Table 3. Silent bugs discovered in PyTorch by TransFuzz and their corresponding bug transfer paths. Underlined issue IDs (e.g., torch #152299) indicate bugs involving API interactions. Note: A single issue may contain multiple bugs.

Bug Type	Bug API	Source Issue	Bug Found	Num
CPU VS GPU	torch.sparse.log_softmax	torch #38839	torch #152293	1
Eager VS Compiled	torch.Tensor.asinh_ + torch.compile	torch #118267	torch #152299	1
Eager VS Jit	torch.set_default_dtype + torch.jit.script	torch #36369	torch #150607	1
Performance Degradation	torch.functional.pca_lowrank	torch #86234	torch #153534	1
Wrong Save/Reload	torch.ao.nn.quantized.Sigmoid	torch #91877	torch #147818	1
Wrong Displayed Message	torch.utils.data.default_collate	torch #47160	torch #153536	7
	torch.fft.ihfft2 + torch.fft.rfft2	torch #67140	torch #149625	
	torch.fft.ihfftn + torch.fft.rfftn	torch #59127	torch #149625	
	torch.randn + torch.compile	torch #147070	torch #149625	
	torch.Tensor.sum + torch.compile	torch #96484	torch #149625	
	torch.onnx.operators.reshape_from_tensor_shape + torch.compile	torch #94831	torch #149625	
	torch.rand_like + torch.jit.trace	torch #128581	torch #147844	
Functionality Not Working as Expected	torch.tensordot	torch #117067	torch #147846	14
	torch.arange + torch.vmap	torch #102208	torch #152295	
	torch.ones + torch.vmap	torch #102208	torch #152295	
	torch.scalar_tensor + torch.vmap	torch #102208	torch #152295	
	torch.utils._sympy.functions.make_opaque_bitwise_fn	torch #138395	torch #147841	
	torch.jit.trace_module	torch #52217	torch #154478	
	torch.utils.data.default_collate + torch.set_default_dtype	torch #36369	torch #150607	
	torch.cuda.manual_seed + torch.compile	torch #107187	torch #149621	
	torch.cuda.manual_seed_all + torch.compile	torch #107187	torch #149621	
	torch.cuda.random.manual_seed + torch.compile	torch #107187	torch #149621	
	torch.xpu.random.set_rng_state_all + torch.compile	torch #107187	torch #149621	
	torch.xpu.random.manual_seed_all + torch.compile	torch #107187	torch #149621	
	torch.xpu.manual_seed_all + torch.compile	torch #107187	torch #149621	
	torch.autograd.grad_mode._force_original_view_tracking	torch #113359	torch #147849	
Wrong Gradient	torch.nn.functional.hardswish	torch #51438	torch #147801	1
Wrong Outputs	torch.addcmul	torch #98691	torch #152294	4
	torch.add	torch #98691	torch #152294	
	torch.xpu.is_bf16_supported	torch #132303	torch #152301	
	torch._dynamo.utils.is_compile_supported	torch #111179	torch #147826	

for a single instance. These results demonstrate the effectiveness of TransFuzz in uncovering a diverse range of silent bugs through context-aware pattern mining.

To evaluate the complexity of silent bugs found by TransFuzz, we analyzed whether they involve API interactions. The results are shown in Table 3, where bugs with underlined IDs (e.g., torch #152299) indicate that their triggering process involves API interactions. While existing tools mostly detect simple, standalone API calls, over 50% of the silent bugs identified by TransFuzz involve inter-API interactions, indicating its ability to uncover state-dependent bugs.

The 31 silent bugs found by TransFuzz expose critical risks to PyTorch's real-world reliability. Their consequences fall into four high-impact categories: (i) *Model Failure*: Gradient or output errors silently corrupt training/inference (e.g., `hardswish` grad, torch#147801; `addcmul` outputs, torch#152294). (ii) *Deployment Breakage*: Mode/hardware mismatches cause silent divergence in production (e.g., CPU/GPU `log_softmax`, torch#152293; Eager/Compiled `asinh_`, torch#152299). (iii) *Production Reliability Loss*: Serialization failures, misleading errors, or state corruption break deployment pipelines (e.g., quantized `Sigmoid` reload, torch#147818; seed APIs under `compile`, torch#149621). (iv) *Performance Degradation*: Critical slowdowns undermine scalability (e.g., `pca_lowrank`, torch#153534).

In addition, among the 31 silent bugs, 23 (74%) persisted for over three years, and 30 (96%) could not be detected by CPU-GPU differential testing, demonstrating the inherent difficulty of exposing such bugs and establishing the necessity of customized oracles beyond simple differential checks.

Crash Detection and Cross-Framework Transfer. Encouraged by its success in silent bug detection, we applied TransFuzz to crash detection, a key metric in fuzzing [Li et al. 2025a; Liu et al. 2023; She et al. 2024; Xiao et al. 2023; Xie and She 2025; Zhang et al. 2025a, 2022, 2024]. Table 4 shows the results. TransFuzz successfully uncovered 25 crash bugs from diverse sources, validating its broad applicability. To evaluate generalizability, we transferred the context-aware bug patterns mined from PyTorch to TensorFlow (v2.18.0) and MindSpore (v2.3.0). Silent bug detection requires precise, context-dependent oracles. Due to substantial behavioral divergence across frameworks (e.g., error handling, exception semantics, and execution models), reliably transferring such oracles across frameworks is fundamentally challenging. As the first work to enable fully automated silent-bug testing, we therefore focus on intra-library transfer, where semantic alignment makes oracle construction and validation feasible. For cross-framework settings, we instead consider crash bugs, whose manifestations are more explicit and stable. Furthermore, TransFuzz was not specifically designed for crash detection. Differences in the crash-triggering conditions between PyTorch and TensorFlow APIs may have caused the bug patterns extracted from PyTorch to be less effective when applied to TensorFlow.

Table 4. Crash bugs discovered by TransFuzz with corresponding bug transfers.

Library	Origin Issue	New Bug Report	Num	Library	Origin Issue	New Bug Report	Num
PyTorch	torch #98857	torch #154425	25	MindSpore	torch #117252	mindspore #IBVKM8	19
	torch #63583	torch #154424			torch #117252	mindspore #IBVKM8	
	torch #63583	torch #154424			torch #120803	mindspore #IBWMCY	
	torch #45967	torch #154423			torch #10654	mindspore #IBWMCY	
	torch #21833	torch #154422			torch #21526	mindspore #IBWMCY	
	torch #113141	torch #154420			torch #84170	mindspore #IBWMCY	
	torch #113141	torch #154420			torch #71629	mindspore #IBWMCY	
	torch #105934	torch #154419			torch #118379	mindspore #IBWMCY	
	torch #39708	torch #150836			torch #74438	mindspore #IBWMCY	
	torch #118990	torch #149800			torch #21526	mindspore #IBWMCY	
	torch #35977	torch #149737			torch #28214	mindspore #IBWMCY	
	torch #29367	torch #149737			torch #122171	mindspore #IBWMCY	
	torch #47608	torch #149724			torch #102208	mindspore #IBWMCY	
	torch #56920	torch #149724			torch #1605	mindspore #IBWMCY	
	torch #72106	torch #149724			torch #84144	mindspore #IBWMCY	
	torch #143484	torch #149626			torch #17750	mindspore #IBWMCY	
	torch #45967	torch #149623			torch #105311	mindspore #IBWMCY	
	torch #20529	torch #149622			torch #75568	mindspore #IBWMCY	
	torch #20529	torch #149622			torch #15918	mindspore #IBWMCY	
	torch #85237	torch #147722					
	torch #31472	torch #150835		TensorFlow	torch #4570	tf #90910	4
	torch #69688	torch #149821			torch #4737	tf #90241	
	torch #63583	torch #149820			tf #58270	tf #90915	
	torch #24731	torch #149822			tf #58270	tf #90915	
	torch #140923	torch #149274					

In total, TransFuzz discovers 79 new bugs across the three frameworks, 12 of which have been confirmed as CVEs (see Table 9 in our extended version), with Common Vulnerability Scoring System (CVSS) scores up to 4.8 and the majority classified as MEDIUM severity. Except for CVE-2025-2149, which is a silent bug caused by incorrect save/reload behavior, all others are crash

bugs (e.g., segmentation faults, floating point exceptions). As summarized in Table 3 and Table 4, these results demonstrate that TransFuzz is not only effective in detecting silent bugs within PyTorch but also highly scalable in crash detection. We further summarize the oracle types used by TransFuzz (see Fig. 12 in our extended version), highlighting its ability to automatically design diverse oracles based on bug behavior. We also present an analysis of API transfer relationships, summarized in Appendix B (in our extended version), which reveals that TransFuzz can trigger a diverse set of bug transfers beyond simple parallel APIs by exploiting shared semantic invariants.

Applicability Analysis. To assess whether TransFuzz's approach is broadly applicable (i.e., effective across a wide range of APIs rather than limited to a few), we analyzed the range of APIs it can target in PyTorch. Unlike tools like TitanFuzz, which require users to manually specify target APIs, TransFuzz automatically selects APIs as fuzzing targets by matching functionally similar APIs to those with historical bugs. A key concern is whether this automated strategy restricts the scope of APIs targeted. As shown in Figure 9, we evaluated the number of APIs TransFuzz selected for testing and the distribution of their test frequencies. TransFuzz targeted 2,421 APIs in PyTorch for fuzzing, compared to 1,593 APIs targeted by TitanFuzz, DFUZZ, FuzzGPT (based on user-provided lists). This demonstrates that our method can effectively target a broad range of APIs for testing.

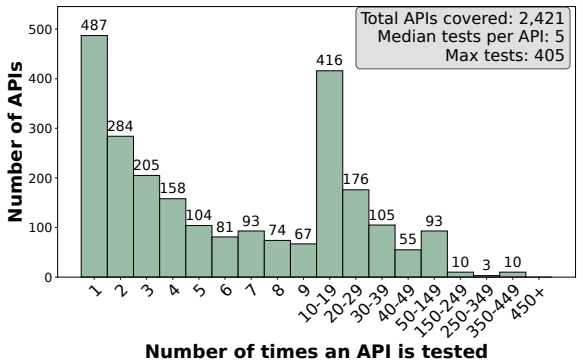


Fig. 9. Frequency distribution of APIs tested by TransFuzz in PyTorch.

4.3 RQ2: Comparison with State-of-the-Art Fuzzers

Baseline Fuzzer Selection. We evaluate TransFuzz against six fuzzers: FreeFuzz [Wei et al. 2022], DeepREL [Deng et al. 2022], ACETest [Shi et al. 2023], TitanFuzz [Deng et al. 2023], DFUZZ [Zhang et al. 2025b], and Pathfinder [Kim et al. 2025]. We selected FreeFuzz, DeepREL, TitanFuzz, and DFUZZ because they represent the state-of-the-art in API-level fuzzing tools. Among these, TitanFuzz [Deng et al. 2023] and DFUZZ [Zhang et al. 2025b] are LLM-based fuzzers, while Pathfinder [Kim et al. 2025] and ACETest [Shi et al. 2023] are specialized fuzzers targeting kernel functions within deep learning libraries. We excluded the following fuzzers from our evaluation: (1) FuzzGPT [Deng et al. 2024]: It is not open-source. (2) FUTURE [Li et al. 2025b]: It lacks adaptation for PyTorch. (3) IvySyn [Christou et al. 2023]: A critical dependency is no longer available, making it impossible to conduct experiments.[2]

Experimental Setup. To ensure a fair comparison, we try to maintain a consistent set of target APIs across all fuzzers. Specifically, we incorporate all relevant APIs listed in Table 3 and Table 4 into each fuzzer's target scope. However, some tools are not fully open-sourced, which limits their

[2]https://github.com/driazati/breakpad

extensibility. For instance, FreeFuzz and DeepREL do not provide their constraint extraction or API matching algorithms; consequently, we are unable to adapt them to test certain APIs.

We primarily evaluate all tools on the latest PyTorch v2.6. However, as Pathfinder is only compatible with PyTorch v2.2, we perform a separate head-to-head comparison between TransFuzz and Pathfinder on the older v2.2 environment to ensure a fair assessment. The results are summarized in Table 5.

Results. The results in Table 5 demonstrate that TransFuzz significantly outperforms existing tools in both bug count and the diversity of bug types.

- Superior Crash Detection: On PyTorch v2.6, TransFuzz identified 25 crashes, which is 3.5× more than the best-performing baseline, ACETest (7 crashes). Other fuzzers like FreeFuzz and TitanFuzz failed to detect any crashes. When compared with Pathfinder on PyTorch v2.2, TransFuzz remains highly competitive. While Pathfinder found 24 crashes (slightly more than TransFuzz's 22), it remains a specialized tool focused solely on crash detection.
- Unique Multi-type Bug Discovery: Unlike baseline fuzzers which are largely confined to detecting crashes or simple CPU/GPU inconsistencies, TransFuzz successfully uncovered a wide array of silent bugs. These include 14 functional errors, 7 wrong error message bugs, and critical cross-mode inconsistencies (e.g., Eager vs. Compiled/JIT), most of which are beyond the reach of traditional fuzzing oracles.

Table 5. Comparison of TransFuzz and existing tools in terms of bug count and type diversity.

Bug Type	**Tested on Torch 2.6**						**Tested on Torch 2.2**	
	FreeFuzz	**DeepREL**	**ACETest**	**TitanFuzz**	**DFuzz**	**TransFuzz**	**Pathfinder**	**TransFuzz**
Crash	0	6	7	0	8	**25**	**24**	22
CPU VS GPU	1	N/A*	N/A	1	1	**1**	N/A	**1**
Eager VS Compiled	N/A	N/A	N/A	N/A	N/A	**1**	N/A	**1**
Eager VS JIT	N/A	N/A	N/A	N/A	N/A	**1**	N/A	**1**
Performance Degradation	N/A	N/A	N/A	N/A	N/A	**1**	N/A	**1**
Wrong Save/Reload	N/A	N/A	N/A	N/A	N/A	**1**	N/A	**1**
Wrong Displayed Message	N/A	N/A	N/A	N/A	N/A	**7**	N/A	**7**
Functionality Not Working as Expected	N/A	N/A	N/A	N/A	N/A	**14**	N/A	**12**
Wrong Gradient	N/A	N/A	N/A	N/A	N/A	**1**	N/A	**1**
Wrong Outputs	N/A	N/A	N/A	N/A	N/A	**4**	N/A	**3**

* "N/A" indicates that the fuzzer is not applicable for detecting this specific type of bug.

4.4 RQ3: Ablation Study on Self-Validation

To systematically assess the effectiveness of our *LLM-Powered Self-Validation* in identifying real bugs, we conducted experiments using both real-world and curated datasets. Our experiments focused on PyTorch as a representative case.

False Positive Rate Analysis. In the real-world experiment presented in Sec. 4.2, TransFuzz identified 31 previously unknown silent bugs. The detection process yielded a false positive rate of 28.58%. After further analysis, we identified the main reasons for TransFuzz's misjudgments as follows: (1) *Oracle design errors*: Logical flaws in the oracle cause correct behaviors to be misclassified as bugs. (2) *Code migration failures*: The bug characteristics from the original code were not preserved or correctly mapped, leading to failures in reproducing the bug in other APIs. (3) *Lack of information resulting in LLM misjudgment*: Due to insufficient information about the API, TransFuzz fails to develop a deep understanding of its functionality.

While the absolute precision figure of 71.42% may appear moderate in isolation, it is important to contextualize it within the extreme difficulty of automatically identifying and verifying silent bugs, a task for which general-purpose oracles are non-existent. TransFuzz is thus among the *first* to systematically tackle this specific challenge using LLMs in DL library fuzzing. The practical significance of this 71.42% precision lies in its ability to serve as an effective filter.

As shown in Fig. 10, we present a failure case that passes all validations but is ultimately deemed non-bug after in-depth inspection by developers. The original issue reports incorrect gradients produced by *torch.compile* when a function composes multiple *flex_attention* calls, where the compiled gradients deviate from eager execution. Based on this pattern, we transferred the test to *multi_head_attention_forward*, again observing consistent gradient mismatches between eager and compiled executions. The transferred case passed all automated validation stages and was initially labeled as *high priority* by developers due to the apparent gradient inconsistency. However, after several rounds of in-depth discussion and inspection, the developers concluded that the discrepancy arises from acceptable numerical differences introduced by compiler-level optimizations rather than a functional bug. These cases demand expert-level reasoning about compiler internals and floating-point behavior, exceeding the scope of our automated analysis.

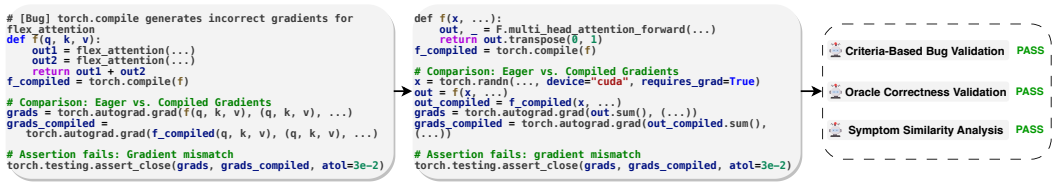
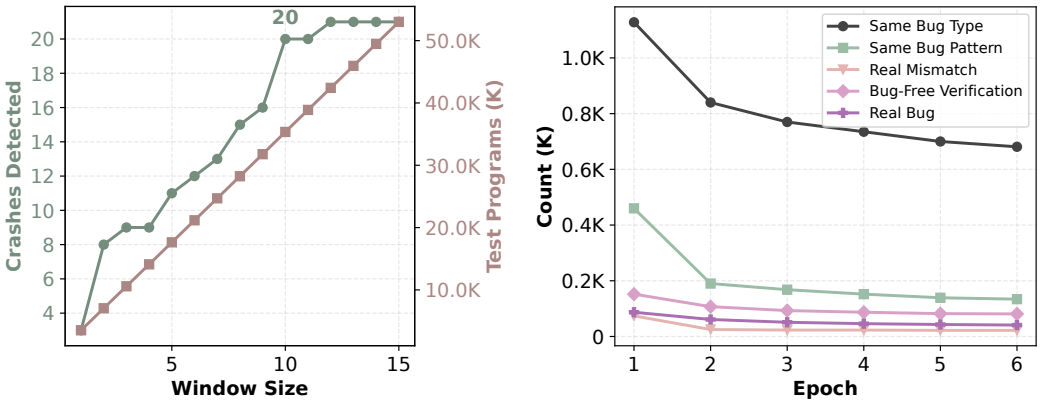


Fig. 10. Failure case passing all automated validations.

Comparison with Pure LLMs. To better understand the contribution of our system design, we conducted an ablation study on a curated dataset (detailed in Appendix C in our extended version). The results show that TransFuzz's structured multi-step validation (achieving 0.8442 accuracy and a 0.7600 F1-score) is markedly superior to simpler LLM-only approaches. This underscores the importance of our system's design, rather than just the model's inherent capabilities. While the real-world complexity and incomplete API documentation pose limitations, the results indicate that TransFuzz's design is key to its practical value in detecting elusive silent bugs.

Different LLMs. To evaluate LLM performance in *LLM-Powered Self-Validation*, we conducted comparative experiments on the curated dataset using leading mainstream models. These include the cost-effective model like GPT-4o mini [gpt 2024], SOTA open-source models (DeepSeek R1 [dee 2025], DeepSeek V3 [dee 2024]), and other commercial models (Doubao-1.5-thinking-pro [dou 2025], Qwen-plus [qwe 2025]). Table 6 summarizes the results of different models on key metrics including accuracy, precision, recall, and F1 score.

As shown in the Table 6, model performance varies considerably. Both Doubao-1.5-thinking-pro and GPT-4.1 mini achieve high accuracy and recall, with F1 scores of 0.7556 and 0.76, respectively, demonstrating strong overall reasoning capabilities suitable for demanding scenarios. DeepSeek R1 leads in precision (0.7692) but shows lower recall. DeepSeek V3 and Qwen-plus display relatively balanced precision and recall. While GPT-4o mini is cost-effective, its low recall (0.2381) and F1 score hinder its suitability for this self-validation task. Strictness varies by model. GPT-4.1 mini adopts more lenient self-validation, achieving high recall (90.48%) but lower precision (65.52%), whereas more conservative models such as DeepSeek R1 enforce stricter validation, resulting in higher precision (76.92%) but substantially lower recall (47.62%). This contrast shows that self-validation



(a) Effect of window size on bug discovery and test program count in PyTorch→PyTorch bug transfer.

(b) Number of cases determined as "pass" under different numbers of repeated checks per prompt.

Fig. 11. Ablation study on hyperparameters.

strictness is directly associated with the recall-precision trade-off: stricter validation leads to higher precision but lower recall, while more lenient validation yields higher recall at the cost of precision.

Table 6. Performance comparison of different LLMs on the LLM-Powered Self-Validation task.

Model	Accuracy	Precision	Recall	F1 Score
GPT-4.1 mini	0.8442	0.6552	0.9048	0.7600
GPT-4o mini	0.7403	0.5556	0.2381	0.3333
DeepSeek R1	0.8182	0.7692	0.4762	0.5882
DeepSeek V3	0.7922	0.6087	0.6667	0.6364
Doubao-1.5-thinking-pro	0.8571	0.7083	0.8095	0.7556
Qwen-plus	0.7922	0.6316	0.5714	0.6000

4.5 RQ4: Ablation Study on Hyperparameters

Window Size. In *Bug Transfer-Guided Test Generation*, the exploration phase adopts a batch testing strategy, selecting 10 untested APIs per round most similar to the original bug API based on the similar API queue. If any new bugs are found in a batch, testing proceeds to the next round; otherwise, it terminates early. To evaluate the impact of window size (i.e., the number of APIs tested per round) on effectiveness and resource usage, we conducted a hyperparameter sensitivity study using the "PyTorch→PyTorch" transfer scenario. For each original buggy API, we attempted transfers to the top 30 most similar target APIs, varying the window size from 1 to 15. For each setting, we recorded both the number of generated test programs and the number of crashes found.

As shown in Fig. 11a, increasing the window size initially yields more crashes, but the benefit plateaus beyond a size of 10. In contrast, test case volume grows linearly with window size. At a window size of 10, 20 out of 21 total crashes are already discovered, while test case count (35,350) remains significantly lower than that at size 15 (53,010). These results suggest that a window size of 10 offers a strong balance between effectiveness and cost, enabling high bug discovery with reduced redundant testing.

Repeated Time. In the *LLM-Powered Self-Validation* process, LLMs are used to automatically analyze test programs and distinguish real bugs from false positives. However, due to the inherent randomness in LLM outputs, repeated prompts can yield inconsistent results, potentially undermining validation stability. To reduce misjudgments caused by randomness, we repeat each check

prompt multiple times. The verdict rule for *LLM-Powered Self-Validation* is: as long as any repetition fails, the case is considered to have failed (i.e., result = epch1_res & epch2_res & epch3_res...). Our goal is to determine how many repetitions should be used for each prompt, so as to minimize misjudgment due to random fluctuations, while also managing the token consumption.

We selected 5 prompts from *LLM-Powered Self-Validation* to filter out false positives for our experiments. For each prompt, we applied 1 to 6 repeats and count the number of cases finally "judged as passed". Fig. 11b shows the results. In the initial stage (1–3 repeats), the descent is the steepest; after 3 repeats, the results enter a stable phase and marginal changes diminish. For example, in prompt *Real Mismatch?*, there is almost no change after the third repetition; in prompt *Same Bug Type?*, the decrease after the third repetition is much smaller; other prompts show the same trend. Therefore, we finally set the repeat number to 3 to maximize the balance between reducing randomness and controlling token consumption.

4.6 RQ5: Ablation Study on Prompt Types

We conducted a manual analysis of the key prompts to evaluate the LLM's performance across various sub-tasks. We randomly selected 10 instances for each prompt. Note that the original issues corresponding to these cases passed the filtering checks of criteria-based bug validation, ensuring that the analyzed issues are of high value.

As shown in Table 7, six tasks, including *Extraction of Bug Determination Criteria*, achieve over 70% accuracy, indicating that LLMs are well-suited for these tasks. However, for the task *Real Bug?*, the accuracy drops to just 40%. We attribute this performance gap to the increased complexity of these tasks, which require LLMs to perform intricate synthesis and reasoning across multiple types of information, making them inherently challenging. To tackle this problem, we adopt a debate mechanism where the LLM questions its initial results from an opposing perspective, ultimately summarizing the findings, which improves precision to 71.42% (see Sec. 3.5). The relatively low accuracy in *Context-Aware Bug Transfer* is due to the fact that the target APIs identified through functionality-based API matching may include APIs with substantial differences. To address this issue, we dynamically adjust the target APIs based on actual fuzzing results (see Sec. 3.4).

Table 7. Performance of LLM on different prompt types.

Prompt Type	Accuracy
Context-Aware Bug Pattern (Fig. 13 in our extended version)	90%
Context-Free Functionality Description (Fig. 14 in our extended version)	100%
Context-Aware Bug Transfer (Fig. 16 in our extended version)	60%
Same Bug Type? (Fig. 20 in our extended version)	90%
Real Mismatch? (Fig. 21 in our extended version)	100%
Same Bug Pattern? (Fig. 22 in our extended version)	70%
Bug-free Verification (Fig. 23 in our extended version)	40%
Extraction of Bug Determination Criteria (Fig. 24 in our extended version)	100%
Real Bug? (Fig. 25 in our extended version)	40%

4.7 RQ6: Cost Analysis

To evaluate the efficiency and cost-effectiveness of TransFuzz, we report the model selection and the cost associated with each key component in our pipeline. Table 8 summarizes the models used and the total expenditure for each submodule throughout the entire evaluation in the "PyTorch→PyTorch" setting.

The *Context-Aware Bug Pattern Extraction* phase accounts for the largest portion of the total cost ($42.16; 47.33%), which is expected given the complexity involved in analyzing a large number of

Table 8. Model selection and cost breakdown for each component.

Component	Model Used	Cost (USD)	Proportion (%)
Context-Aware Bug Pattern Extraction	o3-mini	$42.16	47.33
Functionality-Based API Matching	GPT-4o mini	$0.32	0.003
Bug Transfer-Driven Fuzzing	GPT-4o mini	$27.60	30.98
LLM-Powered Self-Validation	GPT-4.1 mini	$18.99	21.32
Total	–	**$89.07**	**100**

issue reports. This high expenditure is justified by the need for accuracy when extracting context-aware bug patterns from diverse and often lengthy issue descriptions. *Bug Transfer-Driven Fuzzing* is the second most expensive component ($27.60; 30.98%), as it involves generating 35,909 test programs and designing context-aware trigger conditions and oracles for each one. The *LLM-Powered Self-Validation* step, while relatively less costly ($18.99; 21.32%), is crucial for ensuring the correctness of transferred bugs and identifying potential false positives, thereby enhancing the overall reliability of our system. In contrast, *Functionality-Based API Matching* incurs a negligible cost ($0.32; 0.36%) due to the simplicity of the task. Overall, the total monetary cost for running TransFuzz on PyTorch is $89.07.

5 Discussion

TransFuzz leverages triggering contexts and oracles derived from historical bug reports to guide the detection of similar silent bugs. Because silent bugs are highly context-dependent, their detection often requires manual, case-by-case oracle design, making them largely undetectable by existing methods. As the first work to automate silent bug testing in deep learning libraries, our primary goal is to build a practical and effective testing framework rather than exhaustively eliminating all false negatives, which is generally impossible in software testing.

TransFuzz is designed to automatically detect silent bugs while minimizing false positives, a critical bottleneck in existing DL library testing tools. This emphasis inevitably introduces the risk of filtering out some real bugs (i.e., false negatives). To balance this trade-off, we adopt a "transfer-then-verify" strategy. In the transfer stage, we aim for broad coverage to surface as many potential anomalies as possible. In the subsequent validation stage, we apply conservative filtering to remove invalid reports while retaining suspicious cases. Our manual inspection of the misclassified samples in Table 7 shows that the observed errors correspond to misclassifying expected behavior as real bugs (false positives), with no cases where confirmed real bugs were filtered out. While this does not imply that false negatives are fully eliminated, it suggests that the validation is conservative in practice and does not aggressively discard potential bugs. Further optimizations will be explored in future work.

6 Conclusion

In this paper, we presented TransFuzz, a transfer-then-verify framework designed for silent bug fuzzing in deep learning libraries. By leveraging LLMs to extract context-aware bug patterns, match semantically related APIs using functionality-based embeddings, and synthesize tests with customized oracles, TransFuzz enables versatile yet controlled bug transfer. To ensure precision, it integrates an LLM-powered self-validation module to reduce false positives. Evaluations show that TransFuzz uncovers 79 previously unknown bugs across PyTorch, TensorFlow, and MindSpore, demonstrating its effectiveness and cross-framework generalizability.

Data-Availability Statement

The code and datasets used in this work are available at https://github.com/transfuzz/transfuzz. Full experimental results and figures are provided in our [extended version].

Acknowledgments

The HKUST authors were supported in part by a RGC GRF grant under the contract 16214723, an ITF grant under the contract ITS/161/24FP, a HKUST Bridge The Gap fund BGF.001.2025.

References

2024. DeepSeek V3. https://www.deepseek.com.
2024. GPT-4o mini. https://platform.openai.com/docs/models/gpt-4o-mini.
2025. DeepSeek R1. https://www.deepseek.com.
2025. Doubao-1.5-thinking-pro. https://www.volcengine.com/product/doubao.
2025. GPT-4.1 mini. https://platform.openai.com/docs/models/gpt-4.1-mini.
2025. MindSpore. https://gitee.com/mindspore/mindspore.
2025. o3-mini. https://platform.openai.com/docs/models/o3-mini.
2025. PyTorch. http://pytorch.org.
2025. Qwen-plus. https://chat.qwen.ai.
2025. TensorFlow. https://www.tensorflow.org.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. 2023. {IvySyn}: Automated Vulnerability Discovery in Deep Learning Frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2383–2400.

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. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 423–435. doi:10.1145/3597926.3598067

Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 70:1–70:13. doi:10.1145/3597503.3623343

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, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 44–56. doi:10.1145/3540250.3549085

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

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. doi:10.18653/V1/2020.FINDINGS-EMNLP.139

Sorin Mihai Grigorescu, Bogdan Trasnea, Tiberiu T. Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *J. Field Robotics* 37, 3 (2020), 362–386. doi:10.1002/ROB.21918

Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1418–1430. doi:10.1145/3510003.3510092

Shuo Hong, Hailong Sun, Xiang Gao, and Shin Hwei Tan. 2024. Investigating and Detecting Silent Bugs in PyTorch Programs. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*. IEEE, 272–283. doi:10.1109/SANER60148.2024.00035

Sehoon Kim, Yonghyeon Kim, Dahyeon Park, Yuseok Jeon, Jooyong Yi, and Mijung Kim. 2025. Lightweight Concolic Testing via Path-Condition Synthesis for Deep Learning Libraries. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2957–2969. doi:10.1109/ICSE55347.2025.00202

Jialu Li, Haoyu Li, Yuchong Xie, Yanhao Wang, Qinsheng Hou, Libo Chen, Bo Zhang, Shenghong Li, and Zhi Xue. 2025a. Enhancing Real-Time Operating System Security Analysis via Slice-Based Fuzzing. *IEEE Trans. Software Eng.* 51, 12 (2025), 3467–3485. doi:10.1109/TSE.2025.3615642

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 292–303. doi:10.1145/3468264.3468597

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2023a. Contextuality of Code Representation Learning. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 548–559. doi:10.1109/ASE56229.2023.00029

Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023b. Commit-Level, Neural Vulnerability Detection and Assessment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1024–1036. doi:10.1145/3611643.3616346

Zhiyuan Li, Jingzheng Wu, Xiang Ling, Tianyue Luo, Zhiqing Rui, and Yanjun Wu. 2025b. The Seeds of the Future Sprout from History: Fuzzing for Unveiling Vulnerabilities in Prospective Deep-Learning Libraries. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1616–1627. doi:10.1109/ICSE55347.2025.00132

Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. 2021. Computing Systems for Autonomous Driving: State of the Art and Challenges. *IEEE Internet Things J.* 8, 8 (2021), 6469–6486. doi:10.1109/JIOT.2020.3043716

Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. 2023. VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1676–1687. doi:10.1109/ASE56229.2023.00051

Khan Muhammad, Amin Ullah, Jaime Lloret, Javier Del Ser, and Victor Hugo C. de Albuquerque. 2021. Deep Learning for Safe Autonomous Driving: Current Challenges and Future Directions. *IEEE Trans. Intell. Transp. Syst.* 22, 7 (2021), 4316–4336. doi:10.1109/TITS.2020.3032227

Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. 2024. FOX: Coverage-guided Fuzzing as Online Stochastic Control. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 765–779. doi:10.1145/3658644.3670362

Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. 2023. ACETest: Automated Constraint Extraction for Testing Deep Learning Operators. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 690–702. doi:10.1145/3597926.3598088

Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2024. Silent bugs in deep learning frameworks: an empirical study of Keras and TensorFlow. *Empir. Softw. Eng.* 29, 1 (2024), 10. doi:10.1007/S10664-023-10389-6

Wenbo Wang, Tien N. Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2249–2261. doi:10.1109/ICSE48619.2023.00189

Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 995–1007. doi:10.1145/3510003.3510041

Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023. When Less is Enough: Positive and Unlabeled Learning Model for Vulnerability Detection. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 345–357. doi:10.1109/ASE56229.2023.00144

Dongwei Xiao, Zhibo Liu, and Shuai Wang. 2023. PhyFu: Fuzzing Modern Physics Simulation Engines. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1579–1591. doi:10.1109/ASE56229.2023.00054

Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. In *SIGMETRICS/PERFORMANCE '22: ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, Mumbai, India, June 6 - 10, 2022*, D. Manjunath, Jayakrishnan Nair,

Niklas Carlsson, Edith Cohen, and Philippe Robert (Eds.). ACM, 65–66. doi:10.1145/3489048.3522655

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 *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 176–188. doi:10.1145/3533767.3534220

Yuchong Xie and Dongdong She. 2025. HFuzz: Havoc Mode Guided Fuzzing. In *IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT 2025, Ottawa, ON, Canada, April 28-29, 2025*. IEEE, 49–50. doi:10.1109/SBFT66712.2025.00018

Aashish Yadavally, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. (Partial) Program Dependence Learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2501–2513. doi:10.1109/ICSE48619.2023.00209

Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing Automatic Differentiation in Deep-Learning Libraries. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1174–1186. doi:10.1109/ICSE48619.2023.00105

Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025a. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 6999–7018.

Kunpeng Zhang, Shuai Wang, Jitao Han, Xiaogang Zhu, Xian Li, Shaohua Wang, and Sheng Wen. 2025b. Your Fix Is My Exploit: Enabling Comprehensive DL Library API Fuzzing with Large Language Models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 3110–3122. doi:10.1109/ICSE55347.2025.00041

Kunpeng Zhang, Dongwei Xiao, Daoyuan Wu, Shuai Wang, Jiali Zhao, Yuanyi Lin, Tongtong Xu, and Shaohua Wang. 2026. LLM-Powered Silent Bug Fuzzing in Deep Learning Libraries via Versatile and Controlled Bug Transfer. *arXiv preprint arXiv:2602.23065* (2026).

Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. 2022. Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1658–1668. doi:10.1145/3510003.3510063

Kunpeng Zhang, Xiaogang Zhu, Xi Xiao, Minhui Xue, Chao Zhang, and Sheng Wen. 2024. ShapFuzz: Efficient Fuzzing via Shapley-Guided Byte Selection. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society.

Received 2025-10-10; accepted 2026-02-17