CITADEL: Context Similarity Based Deep Learning Framework Bug Finding

XIAOYU ZHANG, Xi'an Jiaotong University, China
JUAN ZHAI, University of Massachusetts, Amherst, United States
SHIQING MA, University of Massachusetts, Amherst, United States
SHIWEI WANG, Xi'an Jiaotong University, China
CHAO SHEN*, Xi'an Jiaotong University, China

With the application of deep learning technology, tools of DL framework testing are in high demand. Existing DL framework testing tools have limited coverage of bug types. For example, they lack the capability of effectively finding performance bugs, which are critical for DL models regarding performance, economics, and the environment. Moreover, existing tools are inefficient, generating hundreds of test cases with few trigger bugs. In this paper, we propose CITADEL, a method that accelerates bug finding in terms of efficiency and effectiveness. We observe that many DL framework bugs are similar due to the similarity of operators and algorithms belonging to the same family. Orthogonal to existing bug-finding tools, CITADEL aims to find new bugs that are similar to reported ones that have known test oracles. CITADEL defines *context similarity* to measure the similarity of DL framework API pairs and automatically generates test cases with oracles for APIs that are similar to the problematic APIs in existing bug reports. CITADEL effectively detects 58 and 66 API bugs on PyTorch and TensorFlow (excluding those rejected by developers or duplicates of prior reports), many of which, e.g., 13 performance bugs, cannot be detected by existing tools. Moreover, 35.40% of test cases generated by CITADEL can trigger bugs significantly transcending the state-of-the-art method (3.90%).

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

Additional Key Words and Phrases: Deep Learning Testing, Deep Learning Library, Software Testing

ACM Reference Format:
Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Shiwei Wang, and Chao Shen. 2025. CITADEL: Context Similarity Based Deep Learning Framework Bug Finding. 1, 1 (October 2025), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the development of Deep Learning (DL) techniques, DL-powered systems are playing an increasingly significant role in software development. For example, Microsoft has developed a new search engine powered by DL techniques to enhance the search results [16]. Moreover, the global AI software market is forecasted to increase from $257 billion in 2025 to $1,459 billion by 2034 [21]. As the backbone of DL-powered systems, DL frameworks (e.g., TensorFlow and

*Chao Shen is the corresponding author.

Authors' addresses: Xiaoyu Zhang, Xi'an Jiaotong University, Xi'an, China, zxy0927@stu.xjtu.edu.cn; Juan Zhai, University of Massachusetts, Amherst, United States, juanzhai@umass.edu; Shiqing Ma, University of Massachusetts, Amherst, United States, shiqingma@umass.edu; Shiwei Wang, Xi'an Jiaotong University, China, shiwei.wang@stu.xjtu.edu.cn; Chao Shen, Xi'an Jiaotong University, China, chaoshen@xjtu.edu.cn.

XXXX-XXXX/2025/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

PyTorch) empower developers by offering API functions to create, train, optimize, and deploy DL-powered systems. These frameworks support diverse domains, providing societal benefits in areas like image recognition [41], self-driving [38], and natural language processing [51]. Similar to traditional software systems, DL frameworks can also have bugs, which can lead to erroneous outputs, increased system overhead, and even crashes for DL-powered systems, thereby jeopardizing user property and personal safety [24], and contributing to energy inefficiency and environmental issues [45, 50, 60]. Consequently, there is a pressing need for tools capable of identifying bugs in DL frameworks.

There are two primary approaches to testing DL frameworks: model-level testing [39, 40, 66, 79] and API-level testing [33, 80]. Model-level testing mutates existing DL models to generate more diverse DL models and employs differential testing methods to compare model execution results across different frameworks for bug detection. In contrast, API-level testing approaches generate test code directly for DL framework API functions, exposing bugs through fuzzing techniques. For example, DocTer [81] conducts fuzzing for DL frameworks by extracting input constraints from API documentation and using these constraints to guide test case generation. DeepREL [33] identifies relational API functions of DL frameworks and 'borrows' test inputs from invoked API functions to test other relational API functions.

Despite these advancements, existing DL testing tools have notable limitations. Firstly, existing testing tools have limited coverage of bug types. For example, they can hardly detect performance bugs that can significantly impact DL model training and inference speed and degrade responsiveness, resulting in energy waste and environmental concerns [25, 45, 50, 60], especially for large DL models like GPT-3 [65]. The performance bug shown in Fig. 2 causes the time overhead to increase to 2.33 times its original value and a substantial carbon footprint. However, current testing methods cannot detect such performance bugs in DL API functions. Secondly, existing bug-finding tools exhibit inefficiencies in generating test cases that trigger bugs. These tools often leverage random walks or heuristic algorithms to generate test cases. However, due to the huge search space of API arguments and inputs, such approaches often generate numerous test cases, but only a small fraction of them trigger actual bugs. For example, DeepREL generates an excess of 330,000 test cases, yet only 1.23% of them have the potential to trigger bugs. Requiring hundreds of test cases to detect a single bug makes the current DL framework testing tools very inefficient in detecting bugs.

To devise an efficient test method capable of identifying various types of bugs, we thoroughly analyze the API functions of PyTorch and TensorFlow and study their reported issues on GitHub. The API functions of these frameworks naturally fall into distinct groups, where API functions within each group execute similar operators and algorithms, exhibiting a tendency for similar bugs. Considering the convolutional operators `torch.nn.Conv1d`, `torch.nn.Conv2d`, and `torch.nn.Conv3d`, each is designed for inputs of different dimensions. Despite their differences, these operators share commonalities such as call lists (e.g., `aten::convolution` in the source code) and the use of the `cudaLaunchKernel` function for GPU computations. Notably, reported issues [6, 7] emphasize that when a bug arises in one convolution operator, other operators within the same group are prone to similar bugs.

Building upon this observation, we propose CITADEL, a tool that accelerates the finding of bugs in terms of efficiency and effectiveness. Orthogonal to existing tools that explore new anomalous behaviors and report bugs, CITADEL aims to uncover new bugs that are similar to reported ones that have known test oracles, regardless of bug types. It leverages reported bugs on one API function to create test cases for its analogous API functions, effectively addressing the aforementioned limitations observed in existing work, and can easily co-work with other testing methods to accelerate bug discovery. Compared with existing tools, which can only detect status bugs and value bugs, CITADEL has better bug type coverage and effectiveness. It has the

capability to detect bugs regardless of their types, such as performance bugs caused by errors in the underlying implementation or optimization, including unexpected time or memory overhead. Moreover, CITADEL is more effective and efficient in test case generation. It adopts the code that has triggered a bug on a problematic API function to create test cases for its analogous API functions. Essentially, it leverages prior knowledge rather than heuristics to explore potential API bugs in the new context, significantly improving the chances of finding bugs. To be specific, we first collect existing bug reports and identify problematic APIs. CITADEL then utilizes both static and dynamic analyses on the DL framework source code and unit test cases for the identification of analogous API functions. In this process, it extracts context information (e.g., APIs' call stacks) to gauge the similarity between API functions, a concept referred to as *context similarity* in CITADEL. For a collected problematic API, CITADEL modifies the bug-triggering code from its bug report to generate new test cases for its analogous API functions. Throughout this process, CITADEL addresses two potential differences between the API functions: differences in arguments and dimensions, if they exist. Finally, CITADEL executes the generated test cases, employing the buggy behavior of the problematic API function as a test oracle to effectively identify potential bugs in the target API function.

Our evaluation demonstrates that CITADEL successfully identified a total of 77 API bugs in PyTorch and 74 in TensorFlow, including 58 and 66 previously unreported bugs, of which 36 and 56 have been confirmed. Additionally, 49 of these bugs are detected by analogous API pairs that existing approaches do not cover. Furthermore, a noteworthy 35.40% of the test cases synthesized by CITADEL expose bugs, significantly surpassing the 0.74%, 1.23%, and 3.90% bug-triggering capacity exhibited by the test cases generated by DocTer, DeepREL, and TitanFuzz, respectively.

Our contributions are:

- We propose *context similarity* as a measurement for functional similarity among DL framework API functions.
- We develop a novel test case generation method for DL frameworks that leverages the knowledge from confirmed API bugs to synthesize new test cases and uncover bugs in analogous API functions, regardless of bug types.
- We develop a prototype CITADEL based on the proposed idea. The experimental results on PyTorch and TensorFlow show that CITADEL detects 58 and 66 previously unknown API bugs, respectively, among which 36 and 56 have been confirmed or fixed by developers after reporting. 35.40% of test cases generated by CITADEL can be used to trigger bugs.

2 BACKGROUND

2.1 DL Framework API Functions and Models

DL Framework APIs. Like traditional software programs, DL frameworks use various API functions to call source code functions and perform operations. Taking PyTorch [64] as an example, its API functions include performing basic matrix operations (e.g., `torch.mul` for multiply operation), calculating loss functions (e.g., `torch.nn.MSELoss` for measuring mean squared error), and building model layers (e.g., `torch.nn.Conv2d` for convolution layers). When calling an API function, users first need to assign values for its required and optional arguments, where the values of required arguments are mandatory to provide, and the optional arguments have their default values in APIs. Then, the API function runs the underlying source code that performs corresponding calculations and operations on the hardware (e.g., CPU and GPU) and obtains tensors, Boolean values, etc. as the result.

DL Models. A DL model is a parameterized function $F_\theta : X \mapsto Y$, where $x \in X$ is an $m-$dimensional input and $y \in Y$ is the corresponding output label. Typically, a DL model is composed of several

connected layers, and an n-layered model can be represented as $F_\theta = l_1 \circ l_2 \circ \cdots \circ l_n$, where l represents a layer and θ is the model weight. Each layer l_i in the model can be constructed by several DL framework API functions. Therefore, a DL model can also be represented as a directed acyclic graph (DAG) in that API functions are nodes, and the returned values of API (e.g., tensors) are edges. Running and training a DL model F on the input-output pairs (x_i, y_i) is essentially calling a series of API functions and passing their outputs based on the topological sorting of its computation graph [33].

2.2 DL Framework Testing

DL framework testing methods construct test cases (e.g., models) to explore abnormal behaviors of DL frameworks and discover bugs Depending on the generated test cases, existing DL framework testing methods can be mainly divided into model-level testing and API-level testing [33, 80].

Model-level testing. These testing methods usually build a large number of models and apply mutation strategies on models to explore the potential bugs of the APIs and layers in the model. To construct test oracles, prior work performs differential testing by building and testing the same model on multiple DL frameworks [39, 40, 66, 79]. CRADLE is one of the first to use this method to test DL framework bugs. Based on Keras [47], which can build and train models on different DL frameworks as backends, it conducts differential testing on three frameworks (i.e., TensorFlow, CNTK, and Theano) and finds 12 bugs. Additionally, Muffin [39] creatively designs the data tracking method to apply differential testing on the training phase of models and finally discovers 39 new bugs. Although the model-level methods obtain outstanding test results, they still have great limitations in applications. Due to limitations imposed by the test model, these methods typically support only a limited subset of API functions related to the models. For instance, existing research [80] reports that LEMON covers only 35 TensorFlow APIs. Furthermore, since the test oracle relies on the implementations of multiple frameworks, inconsistencies detected during testing are often difficult to verify as real bugs, which hampers the effectiveness of the bug detection process [40, 66].

API-level testing. Different from the model-level methods, the API-level framework testing methods do not depend on the implementations of multiple frameworks and have the capability to test abnormal behaviors of more API functions. API-level testing usually extracts API constraints of inputs and arguments based on the documentation or test code and generates test cases based on the fuzzing technique [29, 32, 33, 80, 81, 83, 88]. DocTer [81] analyzes the API document syntax and extracts input constraints. It can generate test cases for three different DL frameworks and find 94 bugs on these frameworks. EAGLE [76] proposes that some APIs have functional equivalence. It designs 16 new DL equivalence rules and detects 25 inconsistencies and bugs on TensorFlow and PyTorch. In addition, DeepREL [33] designs two elaborated equivalence relations and matches API pairs based on these equivalence relations. It considers the output values and status of APIs in a pair as test oracles for each other and detects both crash and inconsistency bugs for over 1,000 PyTorch API functions. TitanFuzz [31] and ∇Fuzz [83] leverage large language models (LLMs) and automatic differentiation to generate test code and implement API-level fuzzing to detect numerical inconsistencies and crashes in DL frameworks. Note that localizing the root cause of APIs' abnormal behaviors is a challenging task, often requiring significant time and effort. Existing methods [33, 81] typically count the number of abnormal behaviors in different APIs (i.e., API bug in this paper) without distinguishing whether they share the same root cause or implementation errors. Following the prior work, Citadel leverages existing bug reports to effectively find and report bugs on analogous API functions without being limited by bug types.

DL framework bugs. DL framework bugs can be mainly divided into three types through symptoms, i.e., status, value, and performance bugs [27]. Status bugs affect the execution status of the

DL API and model, including crashes, segmentation faults, exceptions, etc. Value bugs that are caused by numerical errors in the computation of DL operators include inconsistent outputs and NaN (Not A Number) outputs. Existing framework testing tools focus on the above two types of bugs [33, 40, 81]. Performance bugs refer to those caused by errors in the underlying implementation or optimization, including unexpected time or memory overhead.

2.3 Code Similarity Measurement

Code similarity measurement aims to evaluate the similarity of multiple code blocks and find potential code clones, plagiarism, and refactoring. Existing static approaches proposed methods based on the metrics, texts, and tokens [36, 53, 56, 78]. Researchers also measure code similarity based on Abstract Syntax Trees (ASTs) and graphs (e.g., control flow graphs (CFG)) [26, 85]. In addition, some research proposes the functional similarity between programs from the perspectives of input and output and function calls [55, 72], etc. However, these methods and tools are usually designed for code snippets in one single programming language, but DL framework API functions execute on both Python and C++ source code and involve various wrappers, which poses a challenge in evaluating the similarity between these API functions. Inspired by existing approaches, CITADEL defines and calculates the context similarity to match and test DL framework API functions. CITADEL calculates the similarity of source code blocks from the perspectives of inputs, outputs, and functions. Then it combines the above results with the called functions and traces of different DL framework APIs to match APIs that have a similar functionality and execution context.

3 MOTIVATION

State-of-the-art DL framework testing tools [33, 39, 81] have two major limitations.

• **Existing approaches have limited coverage on bug types. They focus on status and value bugs, lacking the capability of effectively detecting others, e.g., performance bugs.** Existing methods typically utilize the differential testing techniques to construct pseudo test oracles for bug detection. These approaches compare the output results of the same or equivalent API functions on different frameworks/devices to identify status bugs (e.g., crashes) and value bugs (e.g., NaN outputs) [31, 33, 39, 40, 66, 76]. However, they are generally unable to construct test oracles to detect performance bugs due to the difficulty of obtaining test oracles. In addition, although some metamorphic testing methods have successfully identified several performance bugs [80], their effectiveness is constrained by the manually designed metamorphic relations. As a result, they can only test the specific API behaviors (e.g., those related to tensor types) and are unable to identify broader categories of performance bugs (e.g., the `LazyConvTranspose2d` bug in Fig. 2).

• **Existing methods need to generate numerous test cases to trigger a bug, resulting in inefficient testing.** To uncover bugs within DL frameworks, existing work usually leverages random walks or heuristic algorithms to generate test cases and models, exploring potential API behaviors [33, 39, 40, 81]. On the one hand, considering the vast search space of the arguments and inputs of API functions, the random method has a low probability of generating a test case that reveals a bug. On the other hand, the heuristic algorithm (e.g., Genetic Algorithm) typically requires the construction of large populations and multiple generations of mutation to search for bugs, rendering them impractical. Moreover, whether the evaluation of the heuristic algorithm can effectively guide the testing is questionable. Consequently, existing work typically needs to generate hundreds of test cases to uncover a bug, resulting in inefficient testing on the DL framework.

4 DESIGN

We observe that DL framework API functions naturally fall into groups. For example, in PyTorch, while APIs like `Conv1d`, `Conv2d`, and `Conv3d` expose parameter signatures designed to accommodate

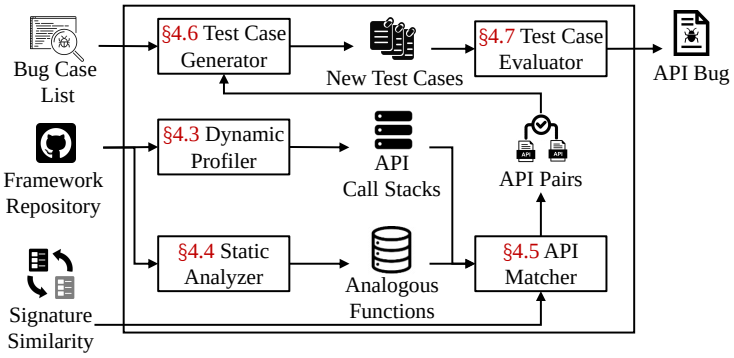
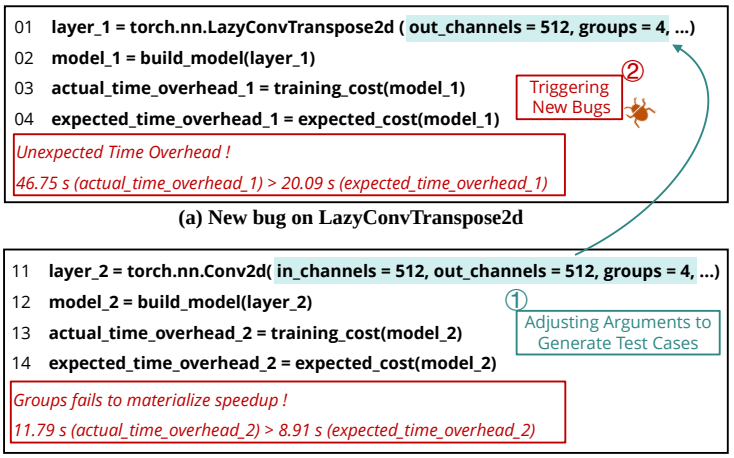


Fig. 1. Overarching Design of CITADEL

different data dimensions, they call similar or even the same operators and algorithms during runtime, indicating that their functionality and implementation are highly similar. Existing research has confirmed that this phenomenon of code cloning is prevalent in DL frameworks [23]. Code cloning not merely significantly increases software maintenance costs but also accelerates the propagation of defects, that is, implementation errors in one API are highly likely to exist in other analogous API functions within the same group [42]. To verify this hypothesis, we sample and manually analyze over 300 real bug reports and corresponding patches from DL frameworks and find that API functions within the same group are highly susceptible to similar or even the same implementation errors, leading to a series of bugs across multiple APIs [2, 4, 6]. For example, prior work [81] has detected a series of crash bugs on the `conv1d`, `conv2d`, and `conv3d` APIs of PyTorch, which actually have the same implementation error (i.e., missing checks on the variable `groups`) and can eventually be fixed by the same patch[1]. Based on the above findings, this paper proposes CITADEL, which aims to systematically exploit the similarities in functionality and implementation between APIs (i.e., contextual similarity) and combine them with known bug reports to efficiently discover bugs in DL framework APIs.

Fig. 1 shows the overview of CITADEL. The inputs to CITADEL include DL framework repositories, real-world bug cases collected from the framework issues, and the signature similarity between DL API functions [33]. Specifically, in this paper, CITADEL is applied to the PyTorch and TensorFlow repositories, two of the most widely used DL frameworks, which have garnered 87K and 189K stars on GitHub, respectively. Bug cases are collected from the issue lists of these two frameworks, including reproducible buggy code and problematic API functions (§4.1). The workflow of CITADEL begins by extracting context information from the DL framework's repository, including the API function call stacks and analogous function groups derived from the framework's source code. In this process, the dynamic profiler (§4.3) generates unit test cases for DL API functions and records their call stacks, capturing the source code functions invoked during execution. Simultaneously, the static analyzer (§4.4) examines the DL framework's source code and clusters analogous functions based on argument and callee similarity. CITADEL proposes *context similarity*, which utilizes API call stacks to measure the similarity between API functions. Additionally, the analogous source code functions are treated as identical during similarity calculation. Leveraging the context similarity and the signature similarity [33], the API matcher (§4.5) matches analogous API pairs. Furthermore, the matcher verifies the arguments of each API pair and discards those with unsolvable argument mismatches. For a problematic API function (i.e., source API) and its analogous API function (i.e.,

[1]https://github.com/pytorch/pytorch/pull/77919



(a) New bug on LazyConvTranspose2d

(b) Confirmed bug on Conv2d

Fig. 2. Performance Bug on `LazyConvTranspose2d`

target API), the test case generator utilizes the reproducible buggy code of the source API, which is collected in the bug case list, to synthesize new test cases for the target API (§4.6). The test case evaluator then executes new test cases and leverages the buggy behavior exhibited by the source API to identify bugs in the target API, including status, value, and performance bugs (§4.7). Finally, CITADEL reports the newly detected API bugs to the user.

CITADEL in an example. CITADEL detects a total of 151 API bugs, including 103 status bugs, 35 value bugs, and 13 performance bugs. Moreover, 35.40% of test cases generated by CITADEL can trigger bugs, and this ratio is only 0.74% and 1.23% in DocTer, DeepREL, respectively.

Here we provide a real-world performance bug found by CITADEL as an example (Lines 1-4 in Fig. 2(a)) to show how it works. One collected bug on `Conv2d` (Fig. 2(b)) reports that the 'groups' argument in this function fails to speed up the training and inference. Grouped convolution aims to employ multiple kernels and produce multiple channel outputs to increase the network efficiency [37, 49]. Therefore, the group convolution is anticipated to bring a lower time overhead compared to executing these convolution layers independently. The code in the collected bug uses the execution time of independent convolution layers to estimate an upper bound on expected time overhead of group convolution, which is accepted by the developers, and finds that the actual overhead of group convolution (11.79 s) is much greater than the expected upper bound (8.91 s), therefore identified the performance bug on `Conv2d`. CITADEL analyzes the context information (e.g., call stacks) of DL API functions to construct pairs of analogous API functions that share context similarity. One such pair consists of `Conv2d` and `LazyConvTranspose2d`. Then, CITADEL generates test cases for `LazyConvTranspose2d` based on the reproducible code of the problematic API `Conv2d`. For each analogous API pair, CITADEL analyzes the arguments of the two API functions to identify differences and makes adjustments to the buggy code accordingly to construct a test case for the target, ensuring the generated test case is executable. In this instance, CITADEL remove 'in_channels' to resolve the difference between APIs' arguments, which is highlighted by green (①). In addition, CITADEL leverages the method in the source bug report of `Conv2d` to estimate the expected upper bound on the time overhead of the grouped `LazyConvTranspose2d` layers using the overhead of independent `LazyConvTranspose2d` layers. The new test case reveals that `LazyConvTranspose2d` with 'group' argument also exhibits a higher time overhead than executing these layers individually, which is the same anomalous behavior as the reported bug in `Conv2d` (②). Specifically, as shown

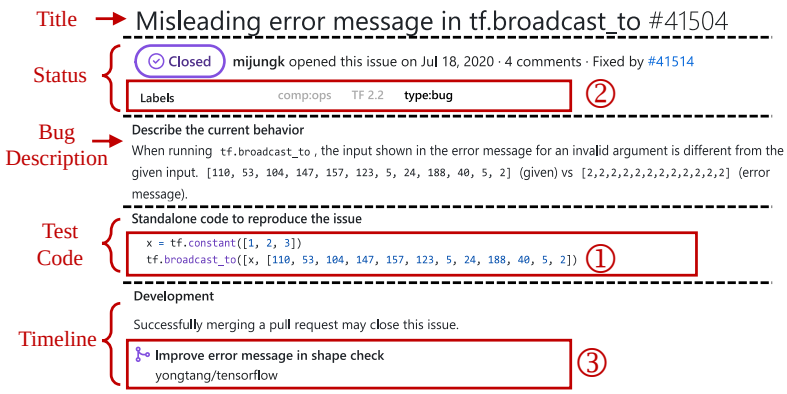


Fig. 3. A Demo of Sampling Bug cases

in Fig. 2(a), when we set the 'out_channels' to 512 and 'groups' to 4 in `LazyConvTranspose2d` layer and construct *model_1* with eight such group convolution layers, the actual time cost of training *model_1* is 46.75 s, which is markedly higher than the time cost of executing these layers individually (20.09 s). Finally, CITADEL discovers a performance bug on the `LazyConvTranspose2d` API function, which has been confirmed by developers [11].

4.1 Preparation: Bug cases Collection

We implement a bug case sampler to extract reproducible bug cases from the latest tens of thousands of GitHub bug issues (both open and closed) of the two DL frameworks as the CITADEL's input, discarding issues that lack reproducible code. Currently, PyTorch and TensorFlow provide well-structured issue templates for bug reporting [14, 18]. These templates request minimal and complete code examples to reproduce the bug and the anomalous behaviors. Taking the `Conv2d` bug in Fig. 2 as an example, its report contains executable code to call the buggy convolution layers (Lines 11-12 in Fig. 2). Additionally, its code calculates the expected time overhead of the group convolution and contrasts it with the actual time cost to directly demonstrate the buggy behavior (Lines 13-14).

Fig. 3 shows a real report from the TensorFlow repository, consisting of the title, status, bug description, test code, and a timeline. To extract the buggy code, the sampler first judges whether one issue includes test code (①) and discards those lacking test code. Then, it checks issue labels (②) and discards reports that are not marked as bugs, crashes, etc. by developers. These issues usually report non-bug problems (e.g., documentation typos) and are not assigned labels by developers. During this process, the sampler also discards reports related to specific hardware (e.g., M1 chips [12]). Limited by the experiment environment, we cannot reproduce these bugs. Finally, the sampler examines the issue timeline (③) and discards issues with fewer than 3 comments, which are usually reports of issues that developers do not care about or intend to work on [9]. It also discards closed issues that lack associated commits or pull requests. Such issues often arise from users' misconceptions of expected behaviors and are promptly addressed by developers [10]. After extracting buggy code, the sampler matches the most frequently mentioned API function from the 'Title' and 'Bug Description' (as shown in Fig. 3) as a problematic API candidate and verifies it in the corresponding test code. If the candidate does not appear in the code, the second most mentioned API is selected, and so on. With such a method, the problematic API in Fig. 3 can be correctly identified as `tf.broadcast`. Detailed implementation of the sampler is in our repository [17].

The bug cases collected by the sampler will be manually verified to determine whether the reported buggy behavior can be reproduced and to classify their bug types (i.e., status, value, or

performance bug). Specifically, we invite two co-authors in the fields of software engineering and artificial intelligence to review the collected cases and label their bug types based on the taxonomy in §2.2. In addition, since CITADEL cannot currently leverage performance bug cases that describe expected overhead in natural language or images [5] to generate new test cases, during the manual review, we only retain those with available code for calculating and estimating the expected overhead. For inconsistent review results, we invite a third co-author to lead the discussion until the review results are recognized by all three.

We acknowledge that, similar to prior approaches, CITADEL also requires a certain amount of manual effort, mainly to verify and ensure the effectiveness of the test cases extracted from bug reports, as described in Section 4.1..

The manual effort at this preparation stage is mainly to verify and ensure the effectiveness of the test cases extracted from bug reports, which will be used in the following experiments (§5.2). Note that in some application scenarios, once the effectiveness of the test cases is ensured, the associated manual effort can be significantly reduced or even eliminated. For example, when CITADEL is used in conjunction with other fuzzing tools, it can directly leverage the test cases that are generated by other tools and have triggered individual API bugs as input. CITADEL can automatically construct new test cases for analogous APIs without requiring additional verification, thereby enabling effective and efficient bug detection.

4.2 Context Similarity

Context similarity calculates the similarity between the runtime context of APIs, and the similar contexts intuitively show that the functionality of APIs would be similar. We observe that on some DL framework APIs (e.g., `Conv1d`, `Conv2d`, and `Conv3d` in PyTorch), although there are differences between their inputs or arguments (e.g., different dimensions), they have similar functionalities and implementations, which can be reflected by context similarity. Many issues and patches [2, 4, 6] in GitHub further reveal that API functions that have similar functionalities are prone to have similar bugs due to one erroneous implementation of an underlying function. Moreover, the prior work [76] has proposed that the functional similarity between API functions can guide the construction of equivalence rules in testing. Therefore, we define and measure the context similarity between API functions to find API functions that have similar functionalities and leverage bugs on one API function to effectively identify potential bugs on its analogous API functions.

Specifically, we measure the context similarity $Sim_{CTX}(A_S, A_T)$ for any API pair (A_S, A_T) in CITADEL:

$$Sim_{CTX}(A_S, A_T) = J(CTX_S, CTX_T),$$

where CTX_S and CTX_T represent the context information of A_S and A_T, which is collected by the static analyzer and dynamic profiler. J indicates the metric to calculate the similarity between CTX_S and CTX_T. In this paper, we use Jaccard similarity coefficient [59] as J to calculate Sim_{CTX}. The greater the similarity between the execution contexts of two API functions, the higher the probability that they have similar underlying implementations and perform similar operators, and they are also more susceptible to similar bugs.

4.3 Dynamic Profiler

The most important context information of APIs is the source code functions they call during execution (i.e., the call stack), which intuitively demonstrates the underlying implementation of the API functions. To effectively collect such context information, the dynamic profiler executes unit test cases for API functions and records call stacks. The unit test cases consist of the test cases collected from DL framework repositories and the test cases generated by existing test case

generation tools [33, 81]. These cases are intended to examine the expected behaviors of APIs during runtime and explore the analogous behaviors and edge cases. Fig. 4 illustrates how CITADEL extract context information and matches `Conv2d` and `LazyConvTranspose2d` in the Fig. 2 as analogous APIs. The solid box in Fig. 4 illustrates part of the API call stacks collected by the profiler. During execution, both APIs call the source code function `aten::convolution` for performing convolution operations and `cudaLaunchKernel` related to GPU services. In addition, `Conv2d` calls the source code function `aten::conv2d` related to the optimization of 2D convolution, while `LazyConvTranspose2d` calls `aten::conv_transpose2d` related to transpose convolution.

4.4 Static Analyzer

Existing research [23] has revealed that a significant number of code blocks and functions with similar implementations exist within the source code of DL frameworks. Source code functions exhibiting similar functionality and implementation patterns may have similar errors, leading to bugs in the DL APIs that invoke them. The static analyzer is designed to identify and cluster these analogous functions within the source code. It then incorporates them as part of the context information provided to the API matcher, facilitating the measurement and matching of APIs with similar execution contexts. Inspired by the prior work [22, 55, 61, 72], the static analyzer determines whether two functions, F_1 and F_2, have similar implementations by evaluating two key aspects: input and output arguments similarity Sim_{io} and callees similarity Sim_{call}.

Input and output arguments play a crucial role in determining functional similarity in code blocks [22, 72]. For a function F_1 within the source code, the static analyzer captures its input and output arguments and formalizes them as a set $IO_{F_1} = \{a_1^1, a_2^1, ..., a_n^1\}$, where a_i^1 represents an input or output argument of F_1. To evaluate the similarity of two sets, we use the Jaccard similarity coefficient [59], a widely utilized statistical measure [67, 73]. The Jaccard similarity coefficient $J(A, B)$ for two given sets A and B is defined as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Then $Sim_{io}(F_1, F_2)$ can be calculated as:

$$Sim_{io}(F_1, F_2) = J(IO_{F_1}, IO_{F_2}) = \frac{|IO_{F_1} \cap IO_{F_2}|}{|IO_{F_1} \cup IO_{F_2}|}$$

Callees, which represent the dependencies of functions, also serve as indicators of functional similarity between code blocks [55, 61]. Similar to input and output arguments, the static analyzer collects and formalizes callees of F_1 as a set $Call_{F_1} = \{f_1^1, f_2^1, ..., f_n^1\}$, where f_i^1 denotes a callee of F_1. CITADEL also computes $Sim_{call}(F_1, F_2)$ through Jaccard similarity coefficient.

$$Sim_{call}(F_1, F_2) = \frac{|Call_{F_1} \cap Call_{F_2}|}{|Call_{F_1} \cup Call_{F_2}|}$$

The static analyzer considers two source code functions to have similar implementations when both similarities exceed the built-in threshold.

$$Sim_{io}(F_1, F_2) \geq \alpha_1 \wedge Sim_{call}(F_1, F_2) \geq \alpha_2,$$

otherwise, they are considered dissimilar. The dashed box in Fig. 4 shows how the static analyzer calculates the similarity between two source code functions `aten::conv2d` and `aten::conv_transpose2d`. Red marks input and output arguments, and blue highlights callees. Given that these two functions share identical input and output arguments (e.g., `const Tensor& weight`) and call the same functions (e.g., `at::convolution`), both Sim_{io} and Sim_{call} exceed the thresholds. Therefore, `aten::conv2d` and `aten::conv_transpose2d` are classified into one group of analogous source code functions.

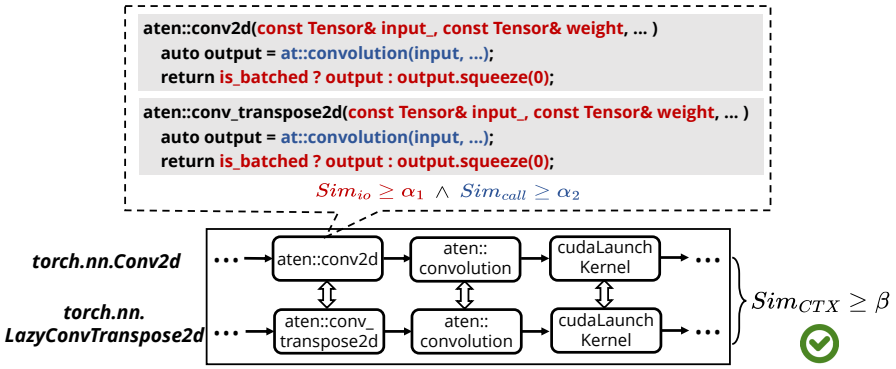


Fig. 4. A Demo Case of Matching APIs Pairs with Context Information

4.5 API matcher

The API matcher first receives context information from the dynamic profiler and static analyzer (i.e., analogous function groups and API call stacks) and matches analogous API pairs based on context similarity and signature similarity [33]. Subsequently, it checks the arguments of analogous API functions and discards the API pairs with unsolvable argument mismatches, which renders the buggy code unusable in test case generation. For instance, the API functions `Conv2d` and `LPPool2d` each have required arguments that are not included in the other, making CITADEL unable to generate a test case for one based on the code of the other, as shown in Fig. 5 (④). Consequently, this API pair is considered to have encountered an argument mismatch and is discarded. Note that matching and filtering API pairs in API matcher is a one-time process. The matched analogous API pairs can be saved and reused in the test case generator.

Matching. For an API function A_S, the call stacks obtained from the dynamic profiler include a set of source code functions it calls during execution. The static analyzer indicates that some of these functions share similar functionality and implementations to other source code functions. The API matcher integrates these two parts of context information to obtain the execution context CTX_S of the API function:

$$CTX_S = \{f'_1, f'_2, ..., f'_m\} \bigcup \{f^S_1, f^S_2, ..., f^S_n\}$$

where f'_i indicates the source code functions in the groups identified in the static analyzer and f^S_j represents other source code functions in the call stack that do not belong to any of the groups. Similarly, we denote the execution context of another API function A_T as CTX_T. Note that CITADEL does not merely compare whether the two API call stacks are the same. When calculating the $Sim_{CTX}(A_S, A_T)$, if both APIs call source code functions from the same group, these functions will be treated as one function because of their similar functionality and implementations. Such a design enables CITADEL to match APIs with similar underlying implementations but different call stacks. The context similarity between A_S and A_T can be calculated by the Jaccard similarity coefficient:

$$Sim_{CTX}(A_S, A_T) = \frac{|CTX_S \cap CTX_T|}{|CTX_S \cup CTX_T|}$$

CITADEL uses the threshold β to evaluate the context similarity between A_S and A_T and considers the two APIs are similar when $Sim_{CTX}(A_S, A_T)$ exceeds β. §5.4 explains the selection of the default value of β in detail.

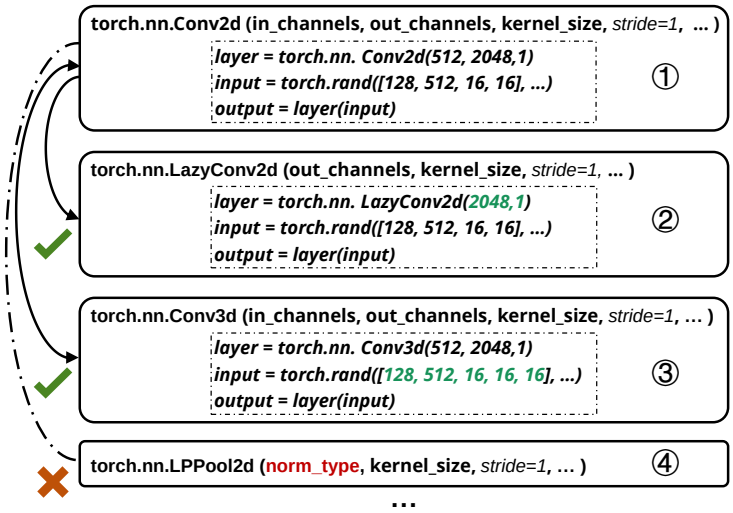


Fig. 5. Verifying API Pairs and Generating Cases

Fig. 4 illustrates a simplified process of matching `Conv2d` and `LazyConvTranspose2d` mentioned in our motivation example via context information. Based on the records of the dynamic profiler (depicted in the solid box), both APIs call the source code functions `aten::convolution` and `cudaLaunchKernel`. For source code functions `aten::conv2d` and `aten::conv_transpose2d`, which are separately called by `Conv2d` and `LazyConvTranspose2d`, they have been divided into one analogous function group by the static analyzer and are considered as the same function when calculating Sim_{CTX}. Note that there are multiple differences in the call stacks of `Conv2d` and `LazyConvTranspose2d`, `aten::conv2d` and `aten::conv_transpose2d` are two demo cases. Since both APIs call almost the same source code functions, their Jaccard similarity coefficients are greater than β, and we match them as a context-similar API pair. In addition to context-similar API pairs, we supplement analogous API pairs matched by the signature similarity calculated by DeepREL [33], which are publicly available. Based on their experiment results, we select the top 20 API functions with the highest signature similarity to one API as its analogous API functions.

Filtering. API matcher checks arguments of matched API functions to avoid argument mismatch problems in the test case generator. For the source API A_S, its arguments P_S can be represented as: $P_S = P_S^r \bigcup P_S^o$, where P_S^r refers to required arguments and P_S^o refers to optional arguments. CITADEL discards the API pair (A_S, A_T), iff A_S and A_T each contain required arguments p_i^r and p_j^r that are not included in the other's arguments set, which means that the test case of either API cannot provide values of required arguments to the other API and generate new test cases.

The discarded API pairs satisfy the following:

$$(\exists p_i^r \in P_S^r, p_i^r \notin P_T) \wedge (\exists p_j^r \in P_T^r, p_j^r \notin P_S)$$

Fig. 5 shows an example of verifying API arguments, where A_S is `torch.nn.Conv2d`(①). Its required argument set P_S^r includes 'in_channels', 'out_channels', and 'kernel_size', and the optional argument set P_S^o contains 'stride' (default value is 1), etc. One of its analogous APIs, `torch.nn.Conv3d`, has the same argument set, allowing the two APIs to generate test cases for each other in the generator and pass the verification(③). `torch.nn.Lazyconv2d` has required arguments 'out_channels' and 'kernel_size'. The test code of `Conv2d` can be modified by removing the first argument to generate test cases for the target API `LazyConv2d`, which also passes the verification(②). Unfortunately, `torch.nn.LPPool2d`

has a required argument 'norm_type' that are not present in `Conv2d`, and `LPPool2d` lacks required arguments 'in_channels' and 'out_channels' (④). Due to the lack of values of required arguments (i.e., 'in_channels', 'out_channels', and 'norm_type'), CITADEL cannot generate new test cases for either API based on the test cases of the other, therefore CITADEL discards the API pair that consists of `LPPool2d` and `Conv2d`.

4.6 Test Case Generator

Given a verified API pair of A_S and A_T, the test case generator synthesizes new test cases C_T for the target API based on the collected buggy code C_S of the problematic API A_S (i.e., source API). As shown in the case of Fig. 2, the generator can adaptively adjust the code of C_T to resolve two kinds of differences (if any) between APIs, namely argument difference and dimension difference, thereby ensuring that the newly generated test cases are executable.

Argument Difference. When the argument set of the source API A_S includes arguments not present in the target API A_T (e.g., `Conv2d` and `LazyConv2d` in Fig. 5), an argument difference arises. To solve this problem, the test case generator modifies the test case C_T by removing irrelevant arguments to make it executable for A_T. The dashed box in Fig. 5 provides an example of resolving arguments difference (②). The argument set of the source API `Conv2d` contains the first argument 'in_channels' that the target API `LazyConv2d` does not have. Therefore, the generator discards the value '512' corresponding to the first argument and keeps only the values '2048' and '1' corresponding to other arguments (marked by green). Additionally, Fig. 2 provides another example that removes the argument 'in_channel=512' in generating test cases for `LazyConvTranspose2d`.

Dimension Difference. As mentioned previously, the DL framework provides a series of APIs for inputs of varying dimensions (e.g, `Conv2d` and `Conv3d`), typically sharing similar implementations and susceptibility to similar bugs [6]. However, the existing methods encounter challenges in constructing test cases for these API functions due to the different dimensions of their arguments [33]. To resolve the dimension difference, the test case generator first obtains the API signatures through open-source libraries (e.g., the 'inspect' library in Python) and identifies the dimension-related arguments from signatures. It then dynamically adjusts the test code by increasing or decreasing the dimensions of argument values based on the dimension information of the API signatures of the source and target APIs, and generates test cases for those API functions. Fig. 5 shows an example of resolving dimension difference and generating available test cases for `Conv3d` (③). The API signature shows that the 'input' of `Conv3d` and `Conv2d` is dimension-related, and the 'input' of `Conv2d` is a 4-dimensional tuple, and the 'input' of `Conv3d` should be a 5-dimensional tuple. The generator recognizes such a dimension difference and expands the 4-dimension tuple received by `Conv2d` to a 5-dimension tuple to adjust the input dimensions (marked in green) and generates executable test cases for `Conv3d`.

CITADEL adaptively generates test cases for various target APIs based on the bug detected in the source API `Conv2d` and finally identifies status bugs on `Conv3d` and `LazyConv2d`, which has been reported to the developers [19].

4.7 Test Case Evaluator

Existing work usually considers the execution results of another API function as a pseudo test oracle and checks whether two API functions produce equal results to detect potential bugs [33, 40, 66]. However, they cannot detect performance bugs due to the difficulty of obtaining test oracles for the runtime overhead of APIs. To effectively identify API bugs regardless of bug types, the evaluator considers the buggy behavior of the source test case C_S as the test oracle and observes whether the new test case C_T has the same buggy behavior. Specifically, the evaluator identifies three types of bugs on A_T as follows.

• *Status bug* arises when C_T throws the identical exception as the source case C_S. The evaluator collects exception details, and if C_T throws the same exception as the original bug case C_S, it is considered that A_T has a status bug.
• *Value bug* arises when C_T generates the same specific numerical errors (e.g., NaN) as C_S. The evaluator logs the outputs of test cases, and if C_T produces anomalous values matching those described in the bug report of A_S, A_T is deemed to have a value bug.
• *Performance bug* arises from an underlying implementation or optimization error, leading to an unexpectedly high overhead on APIs. To detect performance bugs, CITADEL calculates the expected overhead by leveraging the bug report of A_S and records the actual runtime overhead. If C_T exhibits unexpected overhead identical to that described in the bug report of the problematic API A_S (e.g., a time overhead greater than expected), A_T is considered to have a performance bug. Take the case in Fig. 2 as an example, the bug report of `Conv2d` indicates that the performance bug causes the group convolution to exhibit a greater time cost than implementing these convolutional layers individually, which is confirmed by the developers. Leveraging such an anomalous behavior as the test oracle, CITADEL reveals that `LazyConvTranspose2d` also experiences a higher time overhead under the group setting than implementing `LazyConvTranspose2d` layers individually, thereby identifying the performance bug. Please note that this does not imply that `LazyConvTranspose2d` incurs the same cost as `Conv2d` in the original bug report, but rather that both achieve costs exceeding the cost of individual execution (i.e., the expected overhead).

5 EVALUATION

In this section, we aim to answer the following research questions.
RQ1: How effective is CITADEL in detecting real-world bugs?
RQ2: How efficient is CITADEL in detecting real-world bugs?
RQ3: What is the impact of different configurable parameters in CITADEL?

5.1 Setup

Baseline and Metric: We use three state-of-the-art open-sourced testing tools for comparison, namely DocTer [81], DeepREL [33] and TitanFuzz [31]. For the data not shown in their paper (e.g., the number of cases generated by a complete execution), we obtain it by running their open-source code. CITADEL mainly compared with baselines from three metrics:
• *Ratio of test cases that can trigger bugs.* This metric is calculated by dividing the number of cases that can trigger or expose bugs by the total number of generated cases, which can reflect the efficiency of a test approach in generating test cases and detecting bugs. Note that if the baselines do not provide the number of valid cases that can trigger bugs, we will use the number of all bug candidate cases (i.e., the upper bound of valid cases, which may contain a significant number of duplicates) to estimate this metric for the baselines, although this may overestimate the baselines' results on this metric.
• *Average time to detect bugs.* Following the prior work [28, 90], we use the metric of average time to detect bugs to compare the bug detection efficiency of each method. Specifically, we record the time taken by each method to conduct a complete test run and subsequently divide it by the number of detected bugs in the execution. Note that this metric explicitly excludes any preprocessing time, such as API matching in CITADEL and DeepREL or constraint construction in DocTer, since these steps often involve manual effort or produce static results that can be reused in multiple tests. Since the manual cost of verifying which thousands of cases generated by baselines are real bugs is too high, for the baseline method, we directly use the total number of bugs reported in their papers. Note that the fuzzy-based baselines can explore more behaviors and trigger bugs through multiple executions. We actually calculate the upper bound of baselines' performance in this metric, which

Table 1. Summary of Detected API Bugs on PyTorch and TensorFlow

Framework	#Total				#Rejected				#Duplicated				#New				#Confirmed			
	Total	Stat.	Val.	Perf.	Total	Stat.	Val.	Perf.	Total	Stat.	Val.	Perf.	Total	Stat.	Val.	Perf.	Total	Stat.	Val.	Perf.
PyTorch	77	52	15	10	1	0	1	0	18	16	2	0	58	36	12	10	36	21	7	8
TensorFlow	74	51	20	3	2	2	0	0	6	6	0	0	66	43	20	3	56	36	17	3
Total	151	103	35	13	3	2	1	0	24	22	2	0	124	79	32	13	92	57	24	11

is the time of a complete execution divided by the total number of bugs they report. In contrast, CITADEL exploits existing bug reports to discover API bugs without fuzzy generation, and all bugs reported by CITADEL are discovered in one execution.

• *Number of covered APIs.* In prior work, DocTer [81] reports the number of APIs that successfully extract constraints as the number of covered APIs, and DeepREL [33] counts APIs invoked by their 'API_Match_Verifier', which are successfully included in equivalent pairs. Following the prior work, we report the number of APIs matched and verified in our API matcher. In addition, we also count the number of analogous APIs covered by the collected bug cases used in our experiments.

Collected Issues and Context Information. In the experiment, we separately collect and verify 258 and 288 valid bug cases from the latest 30,000 issues of PyTorch and the latest 20,000 issues of TensorFlow, as one input to CITADEL. This process takes approximately 3 weeks per participant. Among them, the problematic APIs in 172 cases (i.e., 104 PyTorch cases and 68 TensorFlow cases) have matched analogous APIs and are used to generate new test cases in the test case generator. These bug cases are publicly available in our repository [17]. In addition, the dynamic profiler constructs test cases for 999 PyTorch APIs and 2,076 TensorFlow APIs and records their call stacks. Based on the experimental results of prior work [56, 72], we set $\alpha_1 = 0.8$ as the threshold of input and output arguments similarity. To strictly judge analogous source code functions and reduce the impact of false positives on subsequent API matching, we set $\alpha_2 = 0.8$ as the threshold for callees similarity. If both the arguments similarity and callees similarity of two functions exceed the thresholds, the two source code functions are judged as analogous functions. The static analyzer separately selects 944 and 7,028 functions in PyTorch and TensorFlow source code that share similarity with at least one other function from the source code and divides them into 2,467 groups. The call stacks and analogous function groups are input into the API matcher as context information.

Software and Hardware: The prototype of CITADEL is implemented on top of Python 3.9. In our experiments, CITADEL test and identify bugs across PyTorch 1.7.0 to 1.13.1 and TensorFlow 2.1.0 to 2.13.0. All experiments are conducted on a server with Intel(R) Xeon(R) Gold 6226R 2.90GHz 16-core processors, 130 GB of RAM, and an NVIDIA 3090 GPU running on Ubuntu 22.04.

5.2 Effectiveness in Detecting Bugs

Experiment Design: To evaluate the effectiveness of CITADEL in detecting real-world bugs, we conduct experiments on the PyTorch and TensorFlow frameworks and report all detected API bugs to developers for confirmation. Our experiment counts the number of API bugs detected by CITADEL, that is, when calling an API with certain inputs triggers one bug, CITADEL will consider that an API bug is detected. In addition, aligned to prior work [81], different inputs triggering the same unexpected behavior on one API will only be counted once. During this process, we record the states of reports, such as confirmed or rejected, and the bug types. Furthermore, following the setting of previous work [33, 80], we collect the number of verified pairs and covered APIs in the API matcher of CITADEL. Note that CITADEL uses the test code in existing bug reports to generate new test cases, and the number of APIs tested in the experiment is related to the collected bug

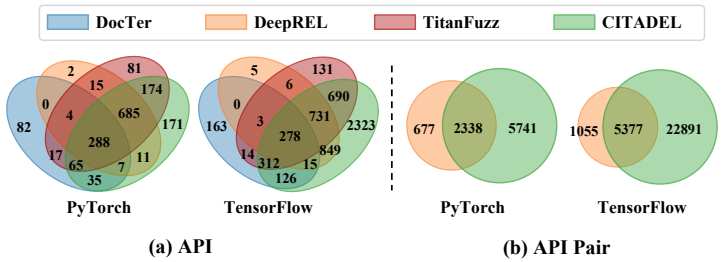


Fig. 6. Comparison of API Coverage and Matched API Pairs

Table 2. Comparison of CITADEL and Baselines Result (Brackets mark the APIs and API Pairs Covered by the Collected Bug Cases)

Approach	Framework	API Coverage		Case Generation			Average Time To Detect Bugs (min)
		#API	#Pairs	#Valid	#Total	Ratio (%)	
DocTer	PyTorch	498	\	45	17,227	0.26	107.16
	TensorFlow	911	\	206	16,632	1.24	25.98
	Total	1,409	\	251	33,859	0.74	41.98
DeepREL	PyTorch	1,071	4,290	2,001	77,662	2.58	40.63
	TensorFlow	1,902	8,808	2,052	252,533	0.81	64.71
	Total	2,973	13,098	4,053	330,195	1.23	58.62
TitanFuzz	PyTorch	1,329	\	2,406	158,185	1.52	43.05
	TensorFlow	2,215	\	11,235	191,862	5.86	101.96
	Total	3,544	\	13,641	350,047	3.90	68.43
CITADEL	PyTorch	1,436 (529)	8,079(797)	82	196	41.84	4.83
	TensorFlow	5,380 (675)	28,268 (1,387)	61	208	29.33	6.61
	Total	6,816 (1,204)	36,347 (2,184)	143	404	35.40	5.70

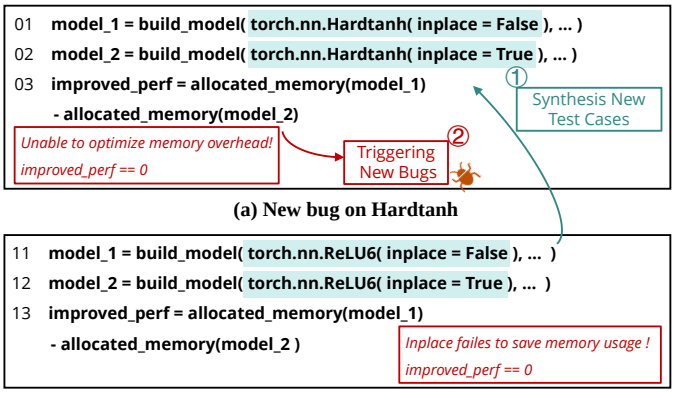
cases. We also report the number of tested APIs using the collected 172 bug cases. How to collect more bug cases to fully utilize the matched API pairs and test more APIs will be a future direction.

Results: Table 1 summarizes the three types of bugs detected by CITADEL, namely 'Stat.', 'Val.', and 'Perf.' (i.e., status, value, and performance bugs). Following the prior work [31, 33, 81], the number of bugs reported here is the number of abnormal behaviors in different APIs (i.e., API bugs), rather than the number of independent implementation defects. The first column of Table 1 displays the DL framework and the following columns indicate the number of all detected API bugs ('#Total'), API bugs that developers do not plan to work on ('#Rejected'), API bugs that have been reported in existing reports ('#Duplicated'), new API bugs that have not been reported ('#New'), new API bugs that have been confirmed ('#Confirmed'). In addition, Table 2 presents a comparison between CITADEL and baselines in the three metrics in §5.1. The first column shows four test approaches in comparison and the second column displays the DL framework. The columns '#API' and '#Pairs' show the number of covered APIs and matched API pairs in each approach, corresponding to the '*Number of covered APIs*' in §5.1. In addition, Fig. 6 uses Venn diagrams to present the comparison of the number of covered APIs and matched API pairs between CITADEL and the baselines. Since we could not find specifics on the APIs and pairs covered by DeepREL in its repository, we use the data collected from its full execution.

Analysis: The results in Table 1 illustrate the effectiveness of CITADEL in detecting various types of real-world bugs. CITADEL generates test cases based on a total of 172 real bugs collected from GitHub repositories and successfully detects 151 API bugs on PyTorch and TensorFlow, out of which only 3 are rejected by developers. These cases show the same anomalous behaviors (e.g., NaN) as the source problematic API, but developers have no plans to fix them. The following 'Rejected Case' provides a detailed analysis of a rejected case. Of the remaining 148 bugs, 24 are duplicates of existing bug reports and the remaining 124 are unreported API bugs, 92 of which have been confirmed by developers. Excluding rejected cases, CITADEL detected 101 status bugs, 34 value bugs, and 13 performance bugs, demonstrating its effectiveness in detecting different types of DL framework bugs. Furthermore, we analyze the patches provided by developers in response to our bug reports and count the number of API bugs that share the same patch as the source API bug. In PyTorch, eight of our issue reports receive official patches (including twelve API bugs), while the patches in six reports indicate that seven newly reported API bugs share the same underlying implementation errors with their corresponding source bugs. In addition, due to the inactivity of the TensorFlow community, none of our bug reports have received patches, making this analysis impossible. This finding highlights a characteristic of our context similarity-based method in CITADEL, i.e., it could identify multiple API bugs caused by the same underlying implementation error. Moreover, compared to other types of bugs, the number of detected performance bugs is relatively small. Our manual analysis of the PyTorch and TensorFlow repositories reveals that the reported performance bugs are infrequent. Take the PyTorch framework as an example, only approximately 300 issues are labeled as 'performance' out of over 10,000 open issues. Moreover, the limited number of issues with reproducible code poses a challenge for CITADEL to gather a significant amount of code related to performance bugs in DL framework repositories. Finally, 8/172 real bugs that CITADEL collects to generate test cases for the matched API pairs are related to performance. Based on these collected performance bugs, CITADEL detects 13 new performance bugs, and 11 of them have been confirmed.

The API matcher of CITADEL covers a total of 1,436 PyTorch APIs and 5,380 TensorFlow APIs, which is 365 more PyTorch APIs and 3,478 more TensorFlow APIs than DeepREL's 'API_Match_-Verifier' and significantly more than 498 PyTorch APIs and 911 TensorFlow APIs covered by DocTer's constraints. Fig. 6 show the comparison of API and API pairs covered by CITADEL and baselines, respectively. In addition, the experiments based on the collected bug cases cover 529 PyTorch APIs and 675 TensorFlow APIs, among which 221 and 338, respectively, are not covered by the baseline methods. CITADEL successfully identifies one PyTorch bug and eight TensorFlow bugs on these previously uncovered APIs. Furthermore, the corresponding test cases cover 797 and 1,387 analogous API pairs on PyTorch and TensorFlow, of which 574 and 1,154 are not covered by the baselines. Note that, due to the limited number of collected bug cases, the experiment currently only uses a small subset of matched analogous APIs and API pairs. Exploring techniques to automatically annotate and efficiently collect a broader set of bug cases to further enhance the detection effect of CITADEL will be our future research direction (§7). The newly covered API pairs enable CITADEL to successfully detect 49 API bugs on PyTorch and TensorFlow, comprising 41 status bugs, 3 value bugs, and 5 performance bugs, which demonstrates the effectiveness of the newly covered APIs and API pairs of CITADEL in detecting bugs. For example, CITADEL newly matches the API pair of `Conv2d` and `LazyConvTranspose2d` via the context similarity and identifies the performance bug in `LazyConvTranspose2d` (Fig. 2). The API pair of `ReLU6` and `HardTanh` in the following case study is also newly covered in CITADEL.

Bug Case 1: In the PyTorch 1.13.1 release, CITADEL detects a performance bug on `Hardtanh` API function, and Fig. 7 presents how CITADEL identify this performance bug. When this bug occurs, the 'inplace' argument in `Hardtanh` can not optimize the memory overhead, and no matter 'inplace'



(a) New bug on Hardtanh

(b) Confirmed bug on ReLU6

Fig. 7. Performance Bug on `Hardtanh`

is assigned as 'True' or 'False', the GPU memory allocated by the model with `Hardtanh` remains constant, as shown in Lines 1-3 in Fig. 7(a). Nevertheless, in the 1.8.0 and 1.9.0 versions, this argument can effectively decrease memory usage (e.g., reduce from 40.43 MB to 21.82 MB). To discover this performance bug, CITADEL first matches `ReLU6` and `Hardtanh` as analogous API functions. Then, based on a collected bug on `ReLU6`, CITADEL synthesizes and executes a test case for `Hardtanh` (①). The bug report of `ReLU6` shows that the model with `ReLU6` allocates the same amount of GPU memory regardless of whether 'inplace' is enabled, and the variable 'improved_perf' in Line 13 of Fig. 7(b) is zero. CITADEL leverages the anomalous behavior of enabling the 'inplace' not decreasing the memory overhead as the test oracle and generates a new test case, as shown in Line 3 of Fig. 7(a). Finally, CITADEL identifies the performance bug on `Hardtanh` (②), and the developers have confirmed this bug and labeled it as 'high priority' [15].

Bug Case 2: CITADEL detects a status bug on the `tensorflow.compat.v1.gather` in TensorFlow 2.14.0 release. When the last dimension of the 'params' argument takes a specific value (e.g., 14), `tf.compat.v1.gather` will crash directly on the GPU without throwing any error message. To detect this status bug, CITADEL first matches `tensorflow.compat.v1.gather` with the source API `tensorflow.raw_ops.Gather` based on context similarity (over 0.9). Subsequently, CITADEL analyzes the arguments of the two API functions and synthesizes a new test case using the collected bug case on the source API for the analogous API `v1.gather`, as shown in Fig. 8 (①). The new test case on `v1.gather` has crashed, exhibiting the same anomalous behavior as observed in `raw_ops.Gather` (②). Finally, CITADEL identifies the new status bug on `v1.gather`. The newly discovered bug has been reported and confirmed by developers [13].

Rejected Case: Although CITADEL effectively identifies API bugs based on whether analogous APIs exhibit the same anomalous behavior as the problematic API, several cases are still rejected by developers. Note that, these rejected cases still exhibit anomalous behaviors, but developers consider them unimportant and have no plans to fix or work on them. Here, we present an example. CITADEL encounters a rejected case when detecting a bug on the `torch.logdet` API. During testing, CITADEL matches `torch.logdet` with `torch.det` and then constructs test cases for the target API `logdet` based on a buggy code of the source API `det`. When executing the test cases, CITADEL finds that both APIs have abnormal and dangerous output value 'NaN' (i.e., not a number) which could further affect subsequent calculations and raise dangerous behaviors [62]. However, developers suggest that the abnormal 'NaN' output on `logdet` is due to the calculation characteristics of this API. When the input matrix is close to being non-invertible (e.g., very small singular values), then

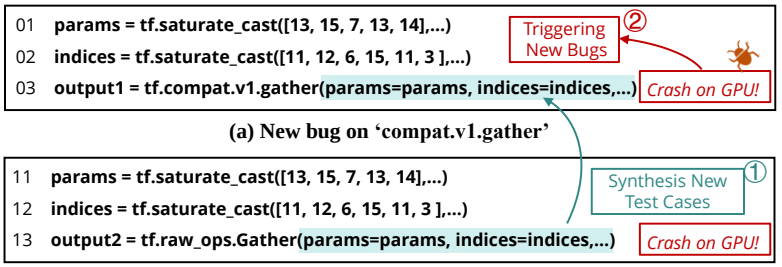


Fig. 8. Crash on `tf.compat.v1.gather`

these APIs may return incorrect results [8]. Rejected cases show the limitations of CITADEL in identifying bugs on APIs that are allowed to exhibit anomalous behaviors. How to identify and filter these unimportant anomalous behaviors to reduce rejected cases will be our future direction.

Answer to RQ1: CITADEL can effectively utilize the bug reports of source APIs to identify API bugs in their analogous API functions. In the experiment, CITADEL leverages 172 bug cases to detect 151 API bugs, among which 124 are newly reported and 13 are performance bugs that existing methods fail to detect. These results demonstrate the effectiveness of CITADEL in finding a wide range of real-world bugs. Furthermore, 49 of these detected bugs originate from analogous API pairs that existing approaches cannot match, while 9 bugs are from APIs that existing methods do not cover. These results highlight the contribution of the APIs and API pairs newly covered by CITADEL in enhancing bug detection.

5.3 Efficiency in Detecting Bugs

Experiment Design and Results: To evaluate the efficiency of CITADEL in generating test cases and triggering bugs, we conduct experiments to calculate and compare the *Ratio of test cases that can trigger bugs* and *Average time to detect bugs* of CITADEL and baseline methods. Specifically, we execute the complete procedure of CITADEL and three baselines to record the total number of generated test cases and the number of test cases that can trigger bugs, and the time cost of testing, as described in §5.1. As the baselines may not save some test cases or inputs, to ensure that we do not mistakenly count such excluded test cases, we directly record the total number of generated test files as the total number of test cases. The experimental results are shown in Table 2. The column '#Valid' shows the number of generated cases that can trigger bugs and the column '#Total' denotes the overall number of generated cases in one complete execution. The column 'Ratio' displays the ratio of the number of cases that triggered bugs to the total number of generated test cases, corresponding to the '*Ratio of test cases that can trigger bugs*' in §5.1. The last column of Table 2 displays the '*Average time to detect bugs*' of each approach.

Analysis: The results in Table 2 demonstrate the efficiency of CITADEL in generating test cases to trigger bugs. In the complete execution of baselines, DocTer generates 33,859 test files on PyTorch and TensorFlow in a complete execution, and only 251 of them (0.74% of the total) are valid cases. The testing process on two frameworks lasts over 99 hours, and the average time to detect bugs in DocTer is 41.98 minutes. On the PyTorch framework, DeepREL spends over 27 hours generating a total of 77,662 test files and marks 2,001 of them as 'can trigger bugs', and the valid test case ratio is 2.58%. DeepREL does not provide the test cases that can trigger bugs or the corresponding numbers on TensorFlow. Therefore, we count the number of candidate bugs it detects (the upper bound of possible bug cases) in Table 2. DeepREL spends over 150 hours testing two frameworks,

and at most 1.23% of all 330,195 generated test files can trigger bugs. The average time to detect bugs of DeepREL is 58.62 minutes. TitanFuzz spends over 72 hours generating 350,047 cases on two DL frameworks, 3.90% of which are candidates and can catch valuable buggy behaviors or trigger bugs. The average time of TitanFuzz to detect bugs is 68.43 minutes. By contrast, Citadel average spends 5.70 minutes to detect one bug, which is only 13.57%, 9.72%, and 8.33% of the average time cost of DocTer, DeepREL, and TitanFuzz, respectively. Citadel is over 10x more time efficient than DeepREL and TitanFuzz in detecting bugs. In testing, Citadel generates a total of 404 test files based on 172 collected bugs within 15 hours, and 143 of them can be used to discover bugs, and the ratio reaches 35.40%. Note that quite a part of the generated test files can be used to detect bugs for multiple analogous API functions at the same time, which improves the efficiency of Citadel in generating test cases and detecting bugs. In addition, our experiments show that the time from when Citadel starts testing to when it triggers the first bug is within 3 minutes, while baselines usually tend to take 8 minutes or more. Citadel, therefore, outranks three baselines in generating and utilizing test cases to detect bugs efficiently.

Existing bug-finding tools use various techniques, such as fuzzing, to explore and discover new anomalous behaviors and detect bugs. As a bug-finding tool orthogonal to existing tools, Citadel aims to use reported bugs to discover API bugs on analogous APIs, therefore, it can be used to enhance the effectiveness and efficiency of bug detection. In the experiment, Citadel leverages 13 bug cases from the baseline method reports and successfully detects 18 API bugs within 3 hours, 11 of which are not reported by the baselines. This demonstrates the potential of collaboration between Citadel and existing bug-finding tools to accelerate DL framework bug detection.

Answer to RQ2: Citadel can efficiently generate test cases to trigger API bugs. In the experiments, 35.40% of the test cases generated by Citadel can trigger bugs, which is over 9.08 times the bug triggering ratio of the baseline methods. Furthermore, the average time to detect bugs of Citadel is 5.7 minutes, while baseline methods need at least 41.98 minutes, demonstrating the efficiency of Citadel in detecting real-world bugs.

5.4 Impacts of Configurable Parameter

Experiment Design and Results: Citadel leverages the built-in threshold β to determine whether two API functions share context similarity. To figure out the impacts of the threshold β and select its default value in Citadel, we conduct experiments with different β values from 0.1 to 0.9 in increments of 0.1. Fig. 9 show the results of these experiments on PyTorch and TensorFlow respectively. The X-axis represents the threshold values. As shown in the legend, the orange and red lines separately indicate the number of API functions covered by context similarity and the number of detected bugs. The blue line illustrates the ratio of the number of target APIs that successfully trigger bugs to the total number of target APIs matched by context similarity (for convenience, we refer to it as *effective target API ratio*). A higher effective target API ratio means that API pairs matched by context similarity are more effective in detecting bugs. Numbers in different colors on the Y-axis show the values of corresponding lines.

Analysis: Fig. 9 illustrates the impact of different values of β on the detection results of Citadel on different frameworks. Since modifying β has similar effects on the testing results of different frameworks, we focus our detailed analysis on the impact of β on PyTorch in this section.

When β is set to 0.6 to 0.8, Citadel can achieve good detection results. A higher β value leads to more stringent evaluations of API pairs sharing context similarity, resulting in a reduction in the number of covered API functions, as shown in Fig. 9(a). Increasing the threshold value from 0.7 to 0.9 results in a sharp decline in the number of covered API functions from 638 to 465. Simultaneously, the bug detection ability of Citadel also declines significantly. When β is

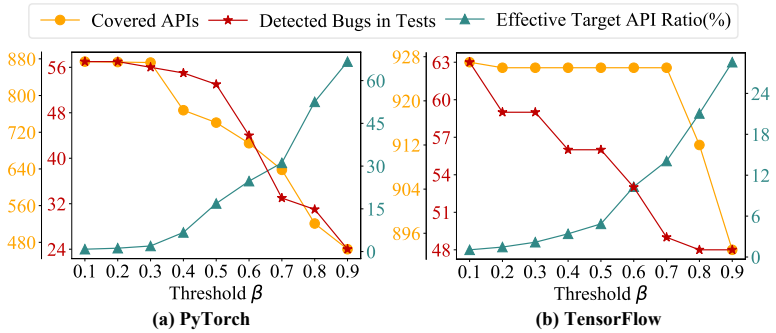


Fig. 9. Impacts of Different β Values on PyTorch and TensorFlow

0.4, CITADEL can detect 55 bugs on PyTorch through context-similar API pairs, and this number decreases to only 24 when β increases to 0.9. However, the increase in the β can improve the effectiveness of matched context-similar API functions in triggering bugs. When the β increases from 0.4 to 0.9, the effective target API ratio on PyTorch rises from 16.87% to 66.67%. Therefore, there is a trade-off between the ratio of effective target API functions and the number of detected bugs and covered API functions. Using a higher threshold can retain API pairs that have more similar contexts, but other API pairs that have the potential to trigger bugs will also be discarded, resulting in a degradation of the overall testing effectiveness. Finally, we set the default value of β to 0.6 on PyTorch and 0.8 on TensorFlow in CITADEL to maximize its API coverage, effectiveness in bug detection, and effective target API ratio.

Answer to RQ3: The configurable parameter β has a significant impact on the bug detection and API coverage of CITADEL. A too large β will discard API pairs that may trigger similar bugs, resulting in a degradation in the effectiveness of bug detection. CITADEL selects $\beta = 0.6$ on PyTorch and $\beta = 0.8$ on TensorFlow to maximize the effectiveness of bug detection.

6 RELATED WORK

In this paper, we propose CITADEL that matches analogous DL framework APIs according to context similarity and argument similarity and generates test cases based on real-world bugs. It is highly related to DL framework testing and code similarity measurement.

DL Framework Testing. Researchers have proposed various methods to test DL framework (e.g., PyTorch and TensorFlow) through model-level methods [39, 58, 77] and API-level methods [33, 76, 81], which have been comprehensively introduced in §2.2. In addition, existing works also design elaborated metamorphic relations to validate the correctness of DL framework implementation [34, 75]. FreeFuzz [80] successfully detects one performance bug using metamorphic testing techniques. However, limited by its metamorphic relation, it can only test framework behaviors related to tensor types and cannot effectively identify other diverse performance bugs (e.g., the `LazyConvTranspose2d` bug in Fig. 2 and the `Hardtanh` bug in Fig. 7). Recently, Zhang et al. [88] propose the test tool 'Predoo', which performs a fine-grained evaluation of the shape variable input and error of 7 operators of the DL framework. Researchers also focus on the security problem of DL frameworks. SkipFuzz [46] uses active learning to learn the input constraints of different library API functions and generates valid test inputs for TensorFlow and PyTorch. It had finally identified 43 crashes on DL frameworks, including 13 CVEs assigned. IvySyn [29] constructs code blocks by DL framework APIs based on a set of offending inputs that trigger memory safety errors in the underlying implementation of DL frameworks (e.g., in C/C++ program language) to trigger security vulnerabilities. In addition,

researchers also detect bugs in other DL underlying libraries such as DL compilers. Liu et al. [52] design a testing method for the ML compiler framework TVM. This method is guided by coverage feedback to mutate the low-level intermediate representation of TVM to achieve more effective fuzzing testing. Shen et al. [70] transfer the knowledge of DL framework fuzzers (e.g., DocTer and DeepRel) to generate effective test cases for diverse DL compiler operators and detect crashes and inconsistencies. Shiri et al. [71] extract corner case patterns from historical issue reports to guide fuzzers in generating test cases and finding bugs more effectively. Their method uses differential testing and crashes to construct test oracles and effectively identifies status and value bugs. However, this design still suffers from the limitation described in §3, namely, the lack of capability of detecting performance bugs. To effectively test rapidly iterating DL frameworks, Xie et al. [82] built upon previous work [76, 81] to design a continuous testing framework for efficiently discovering regression bugs and masked bugs. Recently, Zhang et al. [87] survey the testing methods on various DL libraries and point out the challenges of future testing research.

Different from the above methods, Citadel is built on the concept of code similarity measurement and real-world bugs confirmed by developers. Citadel focuses on DL framework bugs and can leverage existing bug reports on one API to efficiently exploit bugs on its analogous APIs.

Code Similarity Measurement. Existing work has a variety of code similarity measurement and code clone detection methods. These methods can be divided into static and dynamic according to whether the test code needs to be executed. The static methods mainly include metrics-based [36, 63], text-based [53, 68], token-based [69, 74], AST-based [43, 85], and graph-level methods [26, 30]. Among them, the AST and graph-level methods can comprehensively understand the syntax and capture the relationship of the function calls between code blocks, therefore, they have better detection effects but greater overhead than other methods. Existing dynamic methods propose that the functional similarity between programs can be evaluated from the perspectives of input and output [35, 72], abstract memory state [48], etc. Recently, Maertens et al. [54] propose an open-source tool that supports a broad range of programming languages (e.g., C, Java, Python, go). Zhang et al. [89] present an AST-assisted approach for generalizable neural clone detection to find clones in codebases reflecting industry practices. Wang et al. [78] design a code clone detection tool based on the semantic token which enhances the detection capability by complementing the traditional token with semantic information. Our work mainly combines the concept of code similarity measurement into DL framework testing. Citadel matches context-similar APIs based on the similarity of their context information and leverages the bug reports on one API to efficiently find new API bugs on its analogous APIs.

7 DISCUSSION

Discussion. ❶ *Impact of Collected Bug Cases:* Different from existing fuzzing methods [31, 81], which directly generate or mutate test cases for individual APIs, Citadel generates new test cases and finds API bugs for analogous APIs based on collected bug cases. Note that the detection results of Citadel are inherently dependent on the collected bug cases. The more extensive and diverse problematic APIs these cases encompass, the more comprehensive the API coverage Citadel can achieve during detection, leading to the discovery of additional previously unknown bugs. Conversely, a limited number of bug cases concentrated on specific APIs constrain Citadel to testing only a narrow subset of analogous APIs. Although the experiments in §5.2 successfully cover 1,204 DL framework APIs and detect 124 unreported API bugs based on the 172 collected bug cases, demonstrating the effectiveness and efficiency of Citadel, this still represents only a small fraction of the matched analogous APIs, as shown in Table 2. Developing techniques to automatically collect and annotate a more diverse set of bug cases, thereby finding bugs on a broader range of APIs, is a promising future research direction. ❷ *Scope of Citadel:* As a testing

tool orthogonal to existing bug-finding approaches, rather than searching for unknown API bugs from scratch, Citadel leverages a known bug in one API to uncover bugs in its similar APIs. Note that the API bugs discovered by Citadel may stem from the same underlying implementation errors as the known API bugs. However, this does not diminish the value of Citadel. Effectively identifying potential API bugs is crucial for the development and maintenance of DL frameworks. For framework users, Citadel could promptly recognize APIs exhibiting abnormal behavior, thereby preventing users from misusing these buggy APIs and mitigating potential security risks in users' models and software. For framework developers, Citadel efficiently reveals abnormal behaviors across a series of APIs that share similar underlying implementations, providing developers a more comprehensive debugging perspective, helping them infer the root cause and efficiently localize and fix bugs. Moreover, the experiment results in §5.3 demonstrate that Citadel utilizes 13 bug cases reported by baselines to detect 18 API bugs within three hours, 11 of which are previously unreported, demonstrating the efficiency and practicality of Citadel. These findings highlight that Citadel is not a replacement for existing methods, but rather a complement to existing testing tools. It can effectively leverage individual bug reports and improve the effectiveness and efficiency of the entire bug detection and repair process.

Citadel Enhancement. ❶ *Automated Verification and Annotation:* Currently, Citadel receives the bug case list as one of its inputs, which requires manual intervention. How to automatically verify and annotate bug cases (e.g., using code models) is a potential future direction to enhance Citadel's efficiency. ❷ *Performance Bug Reports:* Citadel presently handles performance bug reports that provide code snippets to estimate expected overhead. A valuable future direction would involve automating the calculation of expected overhead or extracting expected behavior from bug reports described in natural language or images [5]. ❸ *Fault Localization:* Some API bugs may share a common root cause, and a single effective patch can potentially resolve multiple related issues [3]. The detection results and corresponding fix patches reported in prior work [1, 81] further support this observation. Note that Citadel, like existing bug-finding tools [33, 81], is designed for bug detection rather than faulty localization or program repair. As such, identifying the root causes of API bugs lies outside the scope of this testing work. We have reported the number of duplicate bugs and the number of API bugs that share the same implementation errors as the source bugs in §5.2. However, since our reports for TensorFlow API bugs currently do not receive any patches, we are unable to conduct the analysis on this framework. We estimate that API bugs with shared underlying implementations also exist in this framework. Therefore, designing error localization methods for automated fault localization remains a valuable direction for future research, which could effectively reduce the debugging workload for DL framework developers (e.g., TensorFlow) and improve bug repair efficiency. ❹ *Code Similarity Measurement:* To mitigate false positives in the static analyzer, Citadel selects strict thresholds to judge analogous source code functions based on the experimental results of prior work [56, 72]. Even if there are still a few false positives in the static analyzer, they can hardly affect the subsequent API matching and bug detection. On the one hand, API functions that are mistakenly matched due to these false positives are easily discarded by the API matcher due to argument mismatches. On the other hand, Citadel identifies bugs when the target API exhibits the same buggy behavior that is consistent with the source API's bug report. Therefore, Citadel can always detect real anomalous behavior. However, the static analyzer could still miss some analogous source code functions. Specifically, different developers may use the same logic to construct functions to solve specific problems, resulting in different implementations of two source code functions but similar functionality [84]. Currently, the static analyzer cannot effectively identify such analogous functions. How to enhance the matching effect of similar source code functions and APIs in Citadel will be a future direction. ❺ *Integration with Existing Tools:* The detection effectiveness of Citadel is influenced by the diversity and quality of

the collected bug cases. Therefore, as described in §4.1, we collect and validate bug cases in the preparation stage to ensure the quality of the bug cases extracted from GitHub repositories. A promising future direction is to integrate Citadel with other bug-finding tools (e.g., fuzzing) into a unified testing pipeline. In such a pipeline, once an API bug is detected by an external tool, the problematic API and its bug-triggering test case can be directly used as input to Citadel. Citadel can then efficiently construct new test cases and detect bugs in analogous APIs. This workflow has the potential to substantially reduce manual effort, enhance the efficiency of bug detection, and assist developers in more effectively identifying the root causes of bugs.

DL Bug Finding. ❶ *LLM Library Bug Finding:* With the advent of Large Language Model (LLM) technology, dependent libraries (e.g., APEX [20]) of LLMs exhibit various bugs, such as crashes during model training [44, 57, 86], which impede the application and deployment of LLMs. However, the substantial runtime overhead of LLMs renders traditional DL fuzz testing methods, which generate millions of test cases, unsuitable for LLM library testing. Developing an efficient LLM library testing method to uncover potential bugs will be a valuable future direction. ❷ *Performance Bug Detection:* Current DL framework testing primarily addresses crashes and numerical inconsistencies, with little attention given to performance bugs that impact model training, economy, and even the environment. Detecting performance bugs necessitates constructing a test oracle and estimating the expected overhead. Our finding suggests that different settings of API arguments may influence the actual overhead (e.g., 'Bug Case 1' of §5.2). Therefore, the runtime overhead of one API function can be qualitatively or even quantitatively estimated based on the description of these arguments, serving as a pseudo-test oracle. Formalizing expected performance changes from API argument descriptions and constructing test oracles to detect performance bugs will be an important future direction.

8 CONCLUSION

This paper presents Citadel, a bug-finding tool for DL frameworks that can find new bugs that are similar to known bugs, regardless of bug types. For a problematic DL framework API function and its associated bugs, it matches analogous API functions from the perspectives of context similarity and signature similarity, and then synthesizes test cases for these analogous API functions. Then, Citadel leverages the behavior of the confirmed bug on the problematic API as the test oracle to evaluate the generated test cases and efficiently identify new API bugs on analogous API functions. Our evaluation on two frameworks shows that Citadel can effectively and efficiently detect status, value, and performance bugs.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments and valuable suggestions. Authors in China are supported partially by the National Key Research and Development Program of China (2023YFE0209800), the National Natural Science Foundation of China (U24B20185, T2442014, 62161160337, 62132011, U21B2018), and the Shaanxi Province Key Industry Innovation Program (2023-ZDLGY-38, 2021ZDLGY01-02). Thanks to the New Cornerstone Science Foundation and the Xplorer Prize.

REFERENCES

[1] 2020. Raise RuntimeError for zero stride pooling. https://github.com/pytorch/pytorch/commit/b0424a895c878cb865947164cb0ce9ce3c2e73ef.

[2] 2021. Workaround for cuFFT bug. https://github.com/pytorch/pytorch/pull/63327.

[3] 2022. Commit 32fbeb1 introduced 7-10x slowdown for th.inverse on large batch matrices. https://github.com/pytorch/pytorch/issues/80735.

[4] 2022. Fix 64bit indexing in vol2col. https://github.com/pytorch/pytorch/pull/87527.

[5] 2022. Significant perf reduction on Python GIL contention with dataloader pinning thread. https://github.com/pytorch/pytorch/issues/77139.

[6] 2022. torch.nn.functional.conv1d/2d/3d crash with floating point exception. https://github.com/pytorch/pytorch/issues/85111.

[7] 2022. Training grouped Conv2D is slow. https://github.com/pytorch/pytorch/issues/70954.

[8] 2023. Extremal values in linalg. https://pytorch.org/docs/master/notes/numerical_accuracy.html.

[9] 2023. GlooBackend leaks file descriptors. https://github.com/pytorch/pytorch/issues/129868.

[10] 2023. Inversing a 0-determinant matrix led to no error. https://github.com/pytorch/pytorch/issues/88680.

[11] 2023. Performance bugs exists in multiple convolution operations(e.g., Convtranspose2d) when useing the groups argument. https://github.com/pytorch/pytorch/issues/95604.

[12] 2023. Performance drops after running tensor multiplication for 15 seconds on M1 MAX (Pytorch MPS). https://github.com/pytorch/pytorch/issues/79402.

[13] 2023. Process aborted when running tf.gather and tf.compat.v1.gather on GPU with large parameters and indices. https://github.com/tensorflow/tensorflow/issues/61780.

[14] 2023. PyTorch Issue Template. https://github.com/pytorch/pytorch/blob/main/.github/ISSUE_TEMPLATE/bug-report.yml.

[15] 2023. Regression bug in torch.nn.ReLU6 and torch.nn.Hardtanh that inplace=True doesn't work in PyTorch 1.10.0 1.13.1. https://github.com/pytorch/pytorch/issues/95432.

[16] 2023. Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web. https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web.

[17] 2023. CITADEL repository. https://github.com/shiningrain/CITADEL.

[18] 2023. TensorFlow Issue Template. https://github.com/tensorflow/tensorflow/blob/master/.github/ISSUE_TEMPLATE/tensorflow_issue_template.yaml.

[19] 2023. torch.nn.Conv2d will segfault when the type of input tensor is float16. https://github.com/pytorch/pytorch/issues/83328#issuecomment-1445337447.

[20] 2024. A PyTorch Extension: Tools for easy mixed precision and distributed training in Pytorch. https://github.com/NVIDIA/apex.

[21] 2025. Artificial Intelligence (AI) Software Market Size, Share, and Trends 2025 to 2034. https://www.precedenceresearch.com/artificial-intelligence-software-market.

[22] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. 2014. Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 225–236.

[23] Maram Assi, Safwat Hassan, and Ying Zou. 2025. Unraveling Code Clone Dynamics in Deep Learning Frameworks. *ACM Transactions on Software Engineering and Methodology* (2025).

[24] Sebastian Blanco. 2023. Report: Tesla Autopilot Involved in 736 Crashes since 2019. https://www.caranddriver.com/news/a44185487/report-tesla-autopilot-crashes-since-2019/.

[25] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. 2003. *Computer systems: a programmer's perspective*. Vol. 2. Prentice Hall Upper Saddle River.

[26] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. 2013. Software plagiarism detection: a graph-based approach. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 1577–1580.

[27] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2022. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology* (2022).

[28] Ruey Long Cheu, Dipti Srinivasan, and Eng Tian Teh. 2003. Support vector machine models for freeway incident detection. In *Proceedings of the 2003 IEEE International Conference on Intelligent Transportation Systems*, Vol. 1. IEEE, 238–243.

[29] 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.

[30] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*. Springer, 37–54.

[31] 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*. 423–435.

[32] 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*. 1–13.

[33] 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*. 44–56.

[34] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. 2017. Validating a deep learning framework by metamorphic testing. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 28–34.

[35] Rochelle Elva and Gary T Leavens. 2012. Semantic clone detection using method ioe-behavior. In *2012 6th International Workshop on Software Clones (IWSC)*. IEEE, 80–81.

[36] Jinan AW Faidhi and Stuart K Robinson. 1987. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education* 11, 1 (1987), 11–19.

[37] Jianhao Gong, Hengyi Li, Qi Li, and Lin Meng. 2021. A Deep Analysis of Grouped Convolution Schemes for Improving Deep Learning Performance.. In *ATAIT*. 45–52.

[38] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.

[39] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 1418–1430.

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

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[42] Judith F Islam, Manishankar Mondal, and Chanchal K Roy. 2016. Bug replication in code clones: An empirical study. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 68–78.

[43] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 55–64.

[44] Weipeng Jiang, Xiaoyu Zhang, Xiaofei Xie, Jiongchi Yu, Yuhan Zhi, Shiqing Ma, and Chao Shen. 2025. The Foundation Cracks: A Comprehensive Study on Bugs and Testing Practices in LLM Libraries. *arXiv preprint arXiv:2506.12320* (2025).

[45] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.

[46] Hong Jin Kang, Pattarakrit Rattanukul, Stefanus Agus Haryono, Truong Giang Nguyen, Chaiyong Ragkhitwetsagul, Corina Pasareanu, and David Lo. 2022. SkipFuzz: Active Learning-based Input Selection for Fuzzing Deep Learning Libraries. *arXiv preprint arXiv:2212.04038* (2022).

[47] Nikhil Ketkar and Nikhil Ketkar. 2017. Introduction to keras. *Deep learning with python: a hands-on introduction* (2017), 97–111.

[48] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*. 301–310.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[50] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. Quantifying the Carbon Emissions of Machine Learning. *arXiv preprint arXiv:1910.09700* (2019).

[51] Hang Li. 2018. Deep learning for natural language processing: advantages and challenges. *National Science Review* 5, 1 (2018), 24–26.

[52] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–26.

[53] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 389–400.

[54] Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, and Bart Mesuere. 2022. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning* 38, 4 (2022), 1046–1061.

[55] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. 2012. Detecting similar software applications. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 364–374.

[56] Md Rakib Hossain Misu and Kazi Sakib. 2017. Interface driven code clone detection. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 747–748.

[57] Yanzhou Mu, Rong Wang, Juan Zhai, Chunrong Fang, Xiang Chen, Jiacong Wu, An Guo, Jiawei Shen, Bingzhuo Li, and Zhenyu Chen. 2025. Designing Deep Learning Frameworks for LLMs: Challenges, Expectations, and Opportunities. *arXiv preprint arXiv:2506.13114* (2025).

[58] Yanzhou Mu, Juan Zhai, Chunrong Fang, Xiang Chen, Zhixiang Cao, Peiran Yang, Kexin Zhao, An Guo, and Zhenyu Chen. 2025. Improving Deep Learning Framework Testing with Model-Level Metamorphic Testing. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 2158–2180.

[59] Allan H Murphy. 1996. The Finley affair: A signal event in the history of forecast verification. *Weather and forecasting* 11, 1 (1996), 3–20.

[60] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 902–912.

[61] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education (TOCE)* 19, 3 (2019), 1–37.

[62] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*. 4901–4911.

[63] Karl J Ottenstein. 1976. An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin* 8, 4 (1976), 30–41.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[65] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350* (2021).

[66] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038.

[67] Chaiyong Ragkhitwetsagul, Jens Krinke, and Bruno Marnette. 2018. A picture is worth a thousand words: Code clone detection based on image similarity. In *2018 IEEE 12th International workshop on software clones (IWSC)*. IEEE, 44–50.

[68] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*. IEEE, 172–181.

[69] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168.

[70] Qingchao Shen, Yongqiang Tian, Haoyang Ma, Junjie Chen, Lili Huang, Ruifeng Fu, Shing-Chi Cheung, and Zan Wang. 2024. A Tale of Two DL Cities: When Library Tests Meet Compiler. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 305–316.

[71] Nima Shiri Harzevili, Mohammad Mahdi Mohajer, Moshi Wei, Hung Viet Pham, and Song Wang. 2025. History-Driven Fuzzing for Deep Learning Libraries. *ACM Transactions on Software Engineering and Methodology* 34, 1 (2025), 1–29.

[72] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *2016 ieee 24th international conference on program comprehension (icpc)*. IEEE, 1–10.

[73] Jeffrey Svajlenko and Chanchal Kumar Roy. 2017. Fast and flexible large-scale clone detection with CloneWorks.. In *ICSE (Companion Volume)*. 27–30.

[74] Zoran Đurić and Dragan Gašević. 2013. A source code similarity system for plagiarism detection. *Comput. J.* 56, 1 (2013), 70–86.

[75] Chaojin Wang, Jian Shen, Chunrong Fang, Xiangsheng Guan, Kaitao Wu, and Jiang Wang. 2020. Accuracy measurement of deep neural network accelerator via metamorphic testing. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 55–61.

[76] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering*. 798–810.

[77] Jiannan Wang, Hung Viet Pham, Qi Li, Lin Tan, Yu Guo, Adnan Aziz, and Erik Meijer. 2024. D 3: Differential Testing of Distributed Deep Learning With Model Generation. *IEEE Transactions on Software Engineering* (2024).

[78] Wenjie Wang, Zihan Deng, Yinxing Xue, and Yun Xu. 2023. Ccstokener: Fast yet accurate code clone detection with semantic token. *Journal of Systems and Software* (2023), 111618.

[79] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.

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

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

[82] Danning Xie, Jiannan Wang, Hung Viet Pham, Lin Tan, Yu Guo, Adnan Aziz, and Erik Meijer. 2024. CEDAR: Continuous Testing of Deep Learning Libraries. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 371–382.

[83] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing automatic differentiation in deep-learning libraries. *arXiv preprint arXiv:2302.04351* (2023).

[84] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software* 204 (2023), 111796.

[85] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[86] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[87] Xiaoyu Zhang, Weipeng Jiang, Chao Shen, Qi Li, Qian Wang, Chenhao Lin, and Xiaohong Guan. 2025. Deep Learning Library Testing: Definition, Methods and Challenges. *Comput. Surveys* 57, 7 (2025), 1–37.

[88] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–412.

[89] Yifan Zhang, Junwen Yang, Haoyu Dong, Qingchen Wang, Huajie Shao, Kevin Leach, and Yu Huang. 2022. ASTRO: An AST-Assisted Approach for Generalizable Neural Clone Detection. *arXiv preprint arXiv:2208.08067* (2022).

[90] Nengwen Zhao, Junjie Chen, Zhaoyang Yu, Honglin Wang, Jiesong Li, Bin Qiu, Hongyu Xu, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2021. Identifying bad software changes via multimodal anomaly detection for online service systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 527–539.