CEDAR: Continuous Testing of Deep Learning Libraries

Danning Xie
Purdue University
West Lafayette, USA
xie342@purdue.edu

Jiannan Wang
Purdue University
West Lafayette, USA
wang4524@purdue.edu

Hung Viet Pham †
York University
Toronto, Canada
hvpham@yorku.ca

Lin Tan *
Purdue University
West Lafayette, USA
lintan@purdue.edu

Yu Guo
Meta Inc.
Menlo Park, USA
yuguo@fb.com

Adnan Aziz
Meta Inc.
Menlo Park, USA
adnanaziz@fb.com

Erik Meijer
Meta Inc.
Menlo Park, USA
erikm@fb.com

***Abstract*—Since Deep Learning (DL) libraries undergo rapid development with thousands of lines of code changes daily, they require continuous testing to detect software bugs and ensure code quality.**

In this paper, we explore DL testing approaches in a continuous testing setting. To make it feasible, we present the first continuous testing framework for DL libraries—*CEDAR*—that integrates two state-of-the-art DL testing approaches (DocTer and EAGLE) efficiently to test two popular DL libraries, PyTorch and TensorFlow. Through the application of CEDAR to 20 versions of PyTorch and TensorFlow, CEDAR detects 83 bugs in 140 APIs. Out of the 83 bugs, 23 are previously unknown bugs with 21 confirmed or fixed by the developers. The results also show CEDAR has effectively shortened the bug detection latency by almost a year (338.6 days) on average. In addition, CEDAR demonstrates its effectiveness in detecting new regression bugs and masked bugs. With three optimization strategies, CEDAR reduces the time and space overhead by a factor of 15.4 and 9.7. We share insights and lessons learned from our research, aiming to advance the development of more effective and efficient continuous testing for DL libraries, benefiting both developers and researchers.

***Index Terms*—testing, continuous integration, continuous testing, deep learning**

I. Introduction

Deep Learning (DL) systems have been widely deployed in many domains including self-driving cars [1] and machine translation [2], [3]. As a result, the reliability and security of DL systems are of vital importance. Nevertheless, conventional testing frameworks and methodologies (e.g., AFL [4]) have limited effectiveness in testing DL libraries, primarily due to the absence of oracles [5], [6] or domain-specific knowledge [7], [8]. For instance, APIs from DL libraries require inputs possessing domain-specific properties (e.g., shape) and data structures (e.g., tensor), making traditional fuzzers either inapplicable or ineffective for testing DL libraries. Fortunately, recent research and techniques have focused on testing DL systems [5]–[12]. For example, DocTer [7] conducts fuzz testing on DL libraries (e.g., PyTorch [13] and TensorFlow [14]) by generating test inputs guided by DL-specific constraints (e.g., number of tensor dimensions) extracted from API documentation, detecting numerous bugs in various Application Programming Interfaces (APIs). Another recent technique EAGLE [6] cross-checks the outputs of two equivalent execution graphs to identify inconsistency bugs in DL libraries. Both techniques have been shown effective in discovering bugs in the specific versions of DL libraries evaluated.

DL libraries such as PyTorch and TensorFlow are actively developed. For example, TensorFlow publishes a nightly build every few days and has released 8 official versions in 2023[1]. There were 142,385 lines of code changes to TensorFlow in a month (November 2023)[2]. These rapid changes often introduce new bugs into the DL libraries. Therefore, applying the automated testing tools **continuously** is essential for efficiently detecting and addressing these bugs, ensuring the reliability and performance of these libraries in practical applications.

One way to continuously maintain the quality of software projects is to apply Continuous Integration (CI), where continuous testing provides instant insights into the quality of the newly committed code and catches newly introduced bugs in time. Integrating existing testing techniques in the CI loop with nightly test runs is one natural direction to improve the reliability of actively developed DL libraries. While existing continuous testing solutions have been implemented on platforms such as Chrome [15] and Google [16], they do not integrate cutting-edge DL testing tools, which are shown to be highly effective in testing DL libraries and uncovering new bugs.

In this paper, we fill this gap by integrating state-of-the-art DL-testing tools, DocTer and EAGLE, into the continuous testing process. Every night, a testing run is scheduled on the latest nightly version of each DL library. These runs conduct

†The work was completed when Hung Viet Pham was at the University of Waterloo.
*Corresponding author.

[1]https://pypi.org/project/tensorflow/#history
[2]https://github.com/tensorflow/tensorflow

fuzz testing and differential testing on the DL APIs, and, in the end, report potential bugs (crashes and inconsistencies).

There are challenges in integrating existing DL testing approaches into the continuous testing process. First, to make integration of these tools into the CI loop feasible in practice, our tests must be completed within a reasonable time frame to facilitate timely feedback [15]–[18]. Second, applying DocTer and EAGLE to nightly builds of actively developed DL libraries can be challenging due to potential day-to-day significant changes. Since these tools rely on API documents (e.g., API signatures and constraints) to generate test inputs, significant source code changes can reduce their efficiency if the documents become outdated. Third, it requires manual efforts to investigate, verify, and report bugs to the developers from a large number of crashes reported daily by the tools.

To address the challenges and enhance the performance and effectiveness of the testing process, we present CEDAR: a continuous testing framework for DL libraries that integrates two state-of-the-art DL-testing tools, DocTer and EAGLE, along with three optimization strategies tailored specifically for integrating these tools. For example, test case reduction, i.e., let EAGLE reuse selected test cases generated by DocTer, reduces the time overhead by a factor of 15.4.

This paper makes the following contributions:

- We built a *continuous* testing framework CEDAR, which integrates two state-of-the-art deep-learning testing approaches (DocTer and EAGLE) *efficiently* to test two popular DL libraries, PyTorch and TensorFlow.
- Our evaluation demonstrates CEDAR's effectiveness in continuous testing, detecting 83 bugs in 140 APIs of 20 versions of PyTorch and TensorFlow. Out of the 83 bugs, 23 are previously unknown bugs with 21 confirmed or fixed by the developers. (Section V-A).
- CEDAR's continuous testing reveals its dual benefits, detecting bugs in *newly added code* in nightly and released versions, as well as (hard-to-detect) bugs in the *existing code* (Section V-A & V-D).
- CEDAR's continuous application across 20 versions shortens the bug detection latency of the evaluated libraries by nearly a year (338.6 days) on average, enabling earlier bug detection (Section V-B).
- CEDAR demonstrates its effectiveness in regression testing and continuous testing by detecting new *regression bugs* and *masked bugs*, both of which are challenging to detect when testing a single version. (Section V-C).
- To make CEDAR's continuous testing practical, we design and implement three optimization strategies that combine generic and tool-specific approaches to reduce the time and space overhead by a factor of 15.4 and 9.7 (Section V-F).
- We present our main findings, insights, and lessons learned for future developers and researchers (Section VIII).

Availability: We share the tool CEDAR and the bug list in [19].

II. Two evaluated testing tools

Testing API functions of DL libraries (e.g., PyTorch [13] and TensorFlow [14]) is crucial because these libraries are widely used and contain bugs [20]–[23]. Previous works DocTer [7] and EAGLE [6] are two state-of-the-art open-source [24], [25] tools designed for testing DL libraries and have detected many bugs in these DL libraries.

A. DocTer

DocTer is a fuzz testing technique for DL libraries (e.g., PyTorch [13] and TensorFlow [14]) that analyzes API documentation to extract DL-specific input constraints and uses them to guide input generation for testing DL API functions. DocTer addresses two challenges: (1) extracting DL-specific constraints, and (2) generating valid input following or violating these constraints.

DL libraries' API functions require DL-specific constraints for their input arguments. Generating DL-specific test cases for these API functions is challenging without the knowledge of these constraints or the ability to use them to generate diverse inputs. In addition, existing testing tools are incapable of generating DL-specific data structures like tensors. DocTer addresses these challenges using the following techniques:

(1) DL-specific constraint extraction: DocTer automatically extracts DL-specific constraints from API documentation by deriving rules predicting parameter constraints from parse tree patterns of API descriptions. Given a small set of annotated API function descriptions, DocTer constructs rules and applies them to a larger set of real-world documents, extracting constraints for widely used DL libraries. DocTer extracts four categories of input properties: *structure*, *dtype*, *shape*, and *valid values*.

(2) Constraint-guided DL-specific input generation: DocTer uses these constraints to guide test generation, producing *conforming inputs* (CIs) and *violating inputs* (VIs). CIs test the core functionality of the API function, while VIs test the API function's input validity checking code. DocTer reports bug-triggering inputs that cause severe crashes.

Using these techniques, DocTer extracts tens of thousands of correct constraints automatically from API documentation for 2,415 API functions from popular DL libraries (e.g., PyTorch) and generates DL-specific inputs to effectively detect bugs in the libraries.

B. EAGLE

Detecting bugs, especially non-crash bugs in DL libraries, is challenging due to the complexity of DL API functionalities and the difficulty of determining expected outputs. EAGLE applies differential testing and uses equivalent graphs to test a single DL implementation. Equivalent graphs use different APIs, data types, or optimizations to achieve the same functionality and should produce identical results given the same input.

EAGLE consists of three main steps:

1) Equivalence rule definition: The authors examine API documentation and non-crash bugs in DL libraries, defining 16 equivalence rules in six categories:

a) Optimization: Computational graph optimization is one of the most popular optimizations that are applied in DL systems. This optimization should not alter the output of a DL system. The rules in this category compare the execution of all

API functions, with or without optimization, given the same input.

b) API redundancy: The built-in API redundancy provided by most DL libraries allows the implementation of one API using other APIs. Rules in this category leverage API redundancy to create equivalent graphs.

c) Data structure equivalence: Numerous DL APIs accept different types of data structures as input while preserving their functionalities. Rules in this category construct equivalent graphs with different input data structures and types.

d) Data format equivalence: Similar to rules in the Data structure equivalence category, rules in this category utilize the fact that input data can be provided to deep learning APIs in different formats. By applying data transformation, these rules generate equivalent graphs.

e) Inverse equivalence: Some APIs in DL libraries have inverse functions. Rules in this category apply APIs and their inverse APIs sequentially to produce equivalent graphs.

f) Model evaluation equivalence: A model should exhibit equivalent behaviors regardless of the inference batch size or before and after saving and loading.

2) Equivalent graph construction: EAGLE builds concrete equivalent graphs for inconsistency detection by identifying relevant APIs for the DL library under test and concretizing the rules for each applicable API.

3) Bug detection: EAGLE generates inputs and compares the outputs of the concretized equivalent graphs given the same input. It reports inconsistencies between equivalent graphs. It considers the outputs as equivalent if the difference is below a certain threshold.

EAGLE applied the 16 equivalence rules to popular DL libraries (e.g., PyTorch) and effectively detected many inconsistencies.

III. CEDAR Approach

A. Overview

Fig. 1 shows our continuous testing framework CEDAR for deep learning libraries. It consists of three phases. First, in the *program installation phase* (Section III-B), the scheduler automatically starts the testing process at the scheduled time. Once it is started, the program installer pulls and installs the latest nightly version of the DL library (e.g., PyTorch). After the program is installed, in the *input generation phase* (Section III-C), DocTer and EAGLE generate test inputs for the DL APIs. To make these tools practical for continuous testing, we have optimized DocTer and EAGLE for efficient integration (Section III-E). For clarity, we refer to the DocTer and EAGLE components in CEDAR as C-DocTer and C-EAGLE in this paper. Finally, in the *test input evaluation phase* (Section III-D), the executor executes the generated inputs and reports a list of abnormal behaviors (e.g., crashes and inconsistencies).

For continuous regression testing of actively developed projects, our testing time must be reasonably short for timely feedback [15]–[18]. Thus, we carefully design our testing framework to ensure quick testing time, e.g., with parallelism, test case reduction, and redundancy removal (Section III-E).

B. Program installation

The testing process is scheduled by the job scheduler that starts CEDAR automatically in the background periodically at fixed times, dates, or intervals. Once it is at the scheduled time, the scheduler calls the program installer to pull and install the latest nightly version of the program under test. For example, one could schedule the testing job for PyTorch at 11:59 PM every Friday. The program installer also saves the version and environment information to the log for record and reference.

C. Input generation

In the *input generation phase*, C-DocTer and C-EAGLE generate test inputs for the API functions of the programs under test. The existing work DocTer [7] collects API information (e.g., API signature and parameter descriptions) from API documents. For example, in the document of PyTorch v1.5.0, the API signature specifies that `torch.nn.functional.conv1d` has two required parameters (`input` and `weight`) and four optional parameters (`bias`, `stride`, etc.). DocTer extracts constraints from the document automatically to guide fuzz testing. For example, the parameter `input` should be a 3-dimensional tensor. Any input that is not a 3-dimensional tensor is rejected by the function's input validity check. Such invalid inputs exercise only the input validity checking code, failing to test the core functionality of the API function.

To test the nightly versions that are not documented, CEDAR uses the API information and constraints provided by DocTer from earlier versions, i.e., PyTorch v1.5.0 and TensorFlow v2.1.0. This is feasible since the difference in the document between each version is small: on average, 96% of the parameters' descriptions remain unchanged between subsequent versions. Based on the extracted constraints, C-DocTer generates both conforming inputs (CIs) and violating inputs (VIs) (Section II-A).

For each equivalence rule (Section II-B), C-EAGLE generates a pair of equivalent graphs for each applicable API. For each of these APIs, C-EAGLE takes inputs generated by C-DocTer and then executes the equivalent graphs with them.

D. Test input evaluation

In the *test input evaluation phase*, the executor invokes the target function with the generated input from C-DocTer and C-EAGLE. If a severe failure occurs, CEDAR reports the input as bug-triggering input. Specifically, CEDAR returns those inputs causing a segmentation fault, floating-point exception, abort, and bus error in the C++ backend. Additionally, if an inconsistency is detected between equivalent graphs generated by C-EAGLE, the input is also reported as bug-triggering input.

E. Efficient Continuous Testing

To make continuous testing practical and efficient, we propose three optimization strategies to reduce time and space overhead.

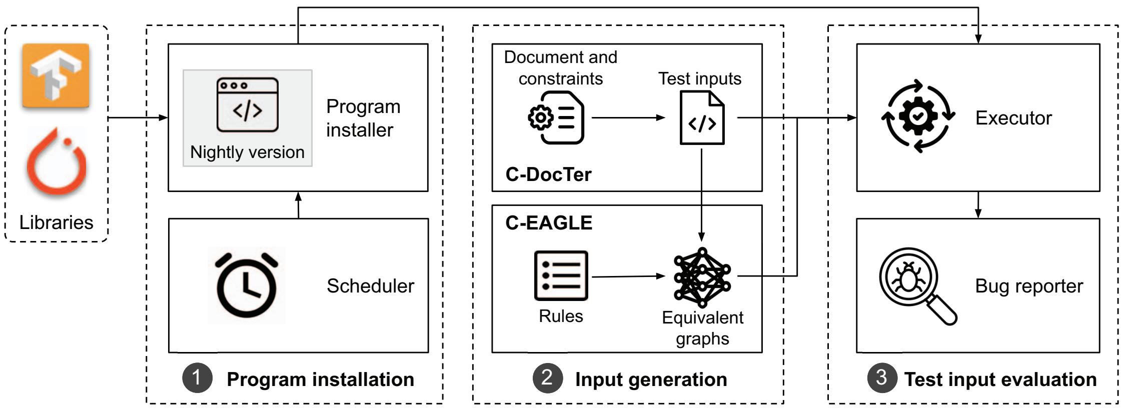


Fig. 1. Overview of CEDAR

a) Parallelism: CEDAR employs parallelism to accelerate the testing procedure. It executes `D_max_proc` C-DocTer processes concurrently, where each process is dedicated to testing a single API. In the case of C-EAGLE, CEDAR initiates `E_max_proc` parallel processes, with each focusing on testing a pair of equivalent graphs against a specific input.

b) Test case reduction: As a standalone testing tool, C-EAGLE relies on C-DocTer's API constraints to generate test inputs to detect inconsistency bugs. To save input fuzzing time, firstly, CEDAR feeds C-EAGLE the inputs generated by C-DocTer directly without having C-EAGLE generate new inputs. Secondly, since C-EAGLE focuses on detecting inconsistency bugs (not crashes), there is little incentive to feed inputs that cause DL API functions to crash to C-EAGLE again, as we have already tested these inputs with C-DocTer to detect crash bugs. Thus, to further accelerate C-EAGLE's input generation and the testing process, C-DocTer generates `D_max_iter` test inputs for each API and only feeds C-EAGLE the first `E_max_iter` inputs that are likely valid (i.e., those that do not make the APIs to crash or throw exceptions) to reduce overhead.

c) Redundancy removal: After CEDAR reports bugs, we keep only the information needed for reproducing the bug (e.g., bug-triggering inputs, status information, etc.) for further investigation and all other space (e.g., inputs and logs) will be freed to reduce overhead.

While *parallelism* is a general strategy, *test case reduction* and *redundancy removal* are tool-specific and tailored specifically for the integration of C-DocTer and C-EAGLE. Our experiments show that the above strategies reduce time and space overhead by 15.4 and 9.7 times respectively (Section V-F).

IV. Experiment Setup

a) Data preparation: As discussed in Section III-C, since nightly versions are not documented, we use the API signatures and constraints extracted by DocTer from earlier versions, i.e., PyTorch v1.5.0 and TensorFlow v2.1.0. DocTer extracts 5,908 and 3,201 constraints from 498 and 911 APIs from PyTorch and TensorFlow, respectively. There are 10 PyTorch APIs and 3 TensorFlow APIs that are no longer supported in the latest nightly versions. CEDAR skips them and generates inputs for the rest 488 and 908 APIs.

C-EAGLE reused the inputs C-DocTer generated for all of the APIs C-DocTer is applied on. Additionally, C-EAGLE generates inputs for 31 PyTorch APIs and 17 TensorFlow APIs whose constraints are not available and are not supported by C-DocTer. For example, the constraints for the PyTorch API `torchvision.transforms.Normalize` is not available since it is from the `Torchvision` package which is out of the scope of DocTer [7], and therefore not available for C-DocTer. In total, C-EAGLE executes 2,461 equivalent graphs (522 for PyTorch and 1,939 for TensorFlow) graphs generated from the 16 equivalence rules (Section II-B) to 519 and 925 PyTorch and TensorFlow APIs respectively.

b) Input generation and testing: During input generation, CEDAR first runs C-DocTer to generate and execute `D_max_iter` CIs and `D_max_iter` VIs for each API and saves the bug-triggering inputs. It also saves the first `E_max_iter` likely-valid inputs for C-EAGLE. After C-DocTer finishes testing all APIs, CEDAR conducts differential testing on equivalent graphs with C-EAGLE. Once C-EAGLE finishes, CEDAR collects and reports crashes and inconsistencies as bugs and frees all other storage that is relevant to the bug-triggering inputs. During testing, CEDAR first runs `D_max_proc` C-DocTer processes in parallel, and then runs `E_max_proc` C-EAGLE processes in parallel (Section III-E).

Specifically, we set `D_max_iter=800`, `E_max_iter=200`, `D_max_proc=20`, and `E_max_proc=24` for PyTorch; and we set `D_max_iter=500`, `E_max_iter=50`, `D_max_proc=6`, and `E_max_worker=24` for TensorFlow. This configuration was chosen to balance computational speed and resource usage efficiently and maintain a reasonable testing duration.

c) Scheduling and versioning: We use cron [26] as a job scheduler which schedules programs periodically at fixed times, dates, or intervals. The testing job is scheduled every day at 11:59 PM. The output, logs, and inputs that are relevant to bugs are stored in the working directory named with the library name, date, and time. For example, the experiment for TensorFlow on November 17, 2023, for testing TensorFlow nightly version 2.16.0.dev20231117 is saved in the folder `tensorflow-11-17-2023-23-59-01`. The version and environment information are logged in the working directory for record and reference.

We run CEDAR on a total of 20 versions, i.e., five nightly versions and five released versions of PyTorch and TensorFlow respectively. The version details are in Section V-A.

d) Apparatus: We use servers with 56 cores and 337G memory and run CEDAR in a Docker container with Ubuntu 18.04.

V. Evaluation and Results

A. Overall bug detection results

We apply our continuous testing framework CEDAR to 20 versions of PyTorch and TensorFlow (five recent nightly versions and five recently released versions per library). After the testing process, CEDAR reports bug-triggering inputs that cause severe failures (e.g., segmentation fault, floating-point exception, abort, and bus error in C++ backend) or result in inconsistencies between equivalent graphs. We then investigate these bug-triggering inputs and report bugs to the developers on GitHub.

We present CEDAR's bug detection results in Table I. In total, CEDAR detects 83 bugs (8 in PyTorch and 75 in TensorFlow) from 140 APIs (35 from PyTorch and 105 from TensorFlow). Out of the 83 bugs, 23 are previously unknown, 21 of which have been confirmed by the developers (15 fixed and 6 confirmed but unfixed yet). Out of the 8 PyTorch bugs that CEDAR detects, 2 bugs, which cause 24 APIs to crash, have been **labeled high priority by PyTorch developers** and are later fixed in the nightly versions. CEDAR also detects 60 $(83 - 23)$ known bugs that are fixed in the subsequent versions or reported by previous approaches (DocTer or EAGLE).

The 83 bugs cause 140 APIs to fail because one bug can cause failures in multiple APIs but is fixed in one location. For example, a bug that C-DocTer detects in PyTorch causes 23 APIs to crash with a segmentation fault. Similarly, C-EAGLE detects a bug in TensorFlow that causes three APIs to produce wrong outputs and results in inconsistencies.

Specifically, C-DocTer detects 77 bugs from 134 APIs in total, including 18 previously unknown bugs. Of the 18 bugs, 17 are confirmed bugs (15 fixed and 2 confirmed but unfixed yet). Among the 59 (77 - 18) known bugs, 53 were detected by DocTer [7]. The other 6 bugs are introduced after the versions on which DocTer was applied, i.e., TensorFlow v2.1.0 and PyTorch v1.5.0. C-EAGLE detects 6 bugs from 6 APIs in total, including 5 previously unknown bugs. The other 1 (6 - 5) bug is a known bug and was previously detected by EAGLE.

We built a continuous testing tool CEDAR for the DL libraries. We learned that CEDAR (both C-DocTer and C-EAGLE) is effective in continuous testing, detecting 83 bugs in 140 APIs of 20 versions of PyTorch and TensorFlow, including 23 previously unknown bugs (21 confirmed or fixed) and 2 high-priority bugs.

We present the breakdown bug detection results for PyTorch in Table II and TensorFlow in Table III. In these tables, we list the PyTorch/TensorFlow versions, the release dates, the total number of bugs detected (Col. *"# Bug Total"*), and the number of bugs detected by C-DocTer (Col. *"# Bug C-DocTer"*) and C-EAGLE (Col. *"# Bug C-EAGLE"*) correspondingly. In the last three columns of these tables, we present the number of *verified new bugs*, *new bugs*, *all bugs*, and *buggy APIs*. A verified bug is fixed or confirmed by the developers.

The number of detected bugs in each version varies for two main reasons. First, CEDAR may uncover more bugs in newer versions as it detects additional bugs within the newly added code. Meanwhile, CEDAR may detect fewer bugs in a newer version when some bugs found in an older version have been fixed, for instance, after we report the bugs. For example, in Table II, the number of detected PyTorch bugs (Col. *"All"*), as well as the number of buggy APIs (Col. *"API"*), decrease in version 1.13.0.dev20220921 compared to the nightly version two days prior, i.e., 1.13.0.dev20220919. This is because two of the bugs that trigger 29 APIs to crash have been fixed by the developers and, therefore, no longer exist in the later versions. Similarly, in Table III, the number of bugs detected in TensorFlow decreases between some versions, i.e., from v2.2.0 to the nightly versions (e.g., 2.11.0.dev20220916), since bugs have been fixed over time. The study of the bugs' lifetime is discussed in Section V-C.

By continuously applying CEDAR, it consistently identifies bugs in two distinct scenarios: (1) within newly added code in both nightly and released versions, and (2) within existing code that has been tested by CEDAR. For instance, consider version 2 of a library introduces an additional 20 lines of code to the existing version 1. If CEDAR is applied to test both version 1 and the modified version 2, it can detect bugs not only in the newly added 20 lines but also in the pre-existing code from version 1. Our findings reveal that *15 of the 83 CEDAR-detected bugs were present in the new code, while 68 were identified in the existing code.* This illustrates CEDAR's capability in promptly detecting bugs in newly integrated code, as well as its effectiveness in re-examining established code to uncover more elusive bugs.

CEDAR's continuous testing reveals its dual benefits, detecting bugs in *newly added code* within both nightly and released versions, as well as (hard-to-detect) bugs in the *existing code*.

B. Reducing bug detection latency

Figure 2 illustrates the bug latency reduction with four crucial time points for a bug denoted as $t_0 - t_3$. Consider the following example: a bug in the TensorFlow API `tf.linalg.diag` was introduced in TensorFlow v2.2.0, released on May 7th, 2020 (t_0). When CEDAR is applied to different TensorFlow

TABLE I

Number of bugs detected by CEDAR. 'Verified', 'New', and 'All' are the number of verified new, new, and all bugs, while 'API' is the number of buggy APIs.

Library	# Bug Total				# Bug C-DocTer				# Bug C-EAGLE			
	Verified	New	All	API	Verified	New	All	API	Verified	New	All	API
PyTorch	6	6	8	35	5	5	7	34	1	1	1	1
TensorFlow	15	17	75	105	12	13	70	100	3	4	5	5
Total	**21**	**23**	**83**	**140**	17	18	77	134	4	5	6	6

TABLE II

Number of bugs detected in PyTorch. 'Verified', 'New', and 'All' are the number of verified new, new, and all bugs, while 'API' is the number of buggy APIs.

PyTorch Version	Release Date	# Bug Total				# Bug C-DocTer				# Bug C-EAGLE			
		Verified	New	All	API	Verified	New	All	API	Verified	New	All	API
1.13.0.dev20220923	09/23/2022	1	1	1	1	1	1	1	1	0	0	0	0
1.13.0.dev20220921	09/21/2022	1	1	1	1	1	1	1	1	0	0	0	0
1.13.0.dev20220919	09/19/2022	5	5	5	27	4	4	4	26	1	1	1	1
1.13.0.dev20220917	09/17/2022	4	4	4	28	4	4	4	28	0	0	0	0
1.13.0.dev20220915	09/15/2022	4	4	4	25	4	4	4	25	0	0	0	0
1.12.1	08/05/2022	5	5	6	27	5	5	6	27	0	0	0	0
1.12.0	06/28/2022	4	4	5	28	3	3	4	27	1	1	1	1
1.11.0	03/10/2022	3	3	4	28	3	3	4	28	0	0	0	0
1.10.1	12/15/2021	2	2	3	8	1	1	2	7	1	1	1	1
1.10.0	10/21/2021	2	2	3	8	1	1	2	7	1	1	1	1
Total		**6**	**6**	**8**	**35**	5	5	7	34	1	1	1	1

TABLE III

Number of bugs detected in TensorFlow. 'Verified', 'New', and 'All' are the number of verified new, new, and all bugs, while 'API' is the number of buggy APIs.

TensorFlow Version	Release Date	# Bug Total				# Bug C-DocTer				# Bug C-EAGLE			
		Verified	New	All	API	Verified	New	All	API	Verified	New	All	API
2.11.0.dev20220927	09/27/2022	5	5	7	8	5	5	7	8	0	0	0	0
2.11.0.dev20220921	09/21/2022	9	10	12	16	7	8	10	14	2	2	2	2
2.11.0.dev20220920	09/20/2022	4	4	5	9	4	4	5	9	0	0	0	0
2.11.0.dev20220918	09/18/2022	8	8	10	15	7	7	9	14	1	1	1	1
2.11.0.dev20220916	09/16/2022	7	8	10	15	7	7	9	14	0	1	1	1
2.10.0	09/06/2022	8	10	12	16	8	9	11	15	0	1	1	1
2.8.0	02/02/2022	8	10	19	25	7	8	17	23	1	2	2	2
2.6.0	08/11/2021	6	7	26	33	6	6	25	32	0	1	1	1
2.4.0	12/14/2020	5	7	44	63	5	6	42	61	0	1	2	2
2.2.0	05/07/2020	2	3	57	82	2	3	56	81	0	0	1	1
Total		**15**	**17**	**75**	**105**	12	13	70	100	3	4	5	5

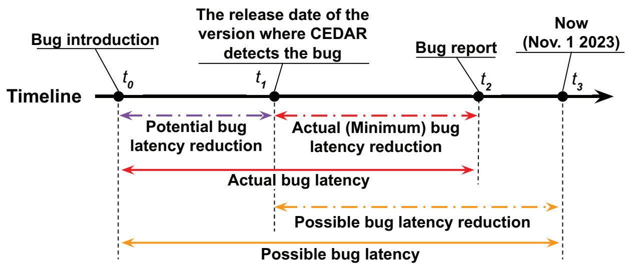

Bug introduction
The release date of the version where CEDAR detects the bug
Bug report
Now (Nov. 1 2023)
Timeline
t_0
t_1
t_2
t_3
Potential bug latency reduction
Actual (Minimum) bug latency reduction
Actual bug latency
Possible bug latency reduction
Possible bug latency

Fig. 2. Bug detection latency illustration.

versions around t_2 on Sep. 15, 2020, it detects the bug in v2.4.0, released on Dec. 14th, 2020 (t_1). We then report the bug to the developers on t_2. t_3 denotes the present.

Initially, we measure the *actual bug latency reduction* (represented in red dashed line in Fig.2) brought about by CEDAR. This is calculated as the time between the release date of the version in which CEDAR detects the bug (t_1) and the date we report the bug to the developers (t_2). In the example provided earlier, if CEDAR had been applied immediately after the release of version v2.4.0 in 2020 (t_1) and detected the bug on the same day, CEDAR could have reduced the bug's latency by $t_2 - t_1$, which is 640 days. In contrast, without CEDAR, this bug would have remained undiscovered until *at least* t_2 (Sep. 15, 2022), i.e., 640 days later, since we are the first to report this bug.

For the 23 previously unknown bugs that CEDAR detects, we compute the *actual (minimum) bug latency reduction*. Our findings indicate that, on average, the minimum bug latency reduction time for these bugs is 338.6 days. Furthermore, under the assumption that the previously unknown bugs detected by CEDAR would have remained undiscovered until today (t_3) if CEDAR had not been applied—-considering the challenge and the effectiveness of employing such tools to detect bugs in DL libraries [6], [7]—-the *possible bug latency reduction* (indicated by the orange dashed line) amounts to 744 days. *In summary, the regular application of CEDAR leads to bugs being detected at least 338.6 days earlier, possibly reaching 744 days, and correspondingly reducing the bug detection latency.*

Second, we investigate the extent to which CEDAR could have reduced the bug latency of a program if developers had applied CEDAR to all nightly versions. We refer to this as the *potential bug latency reduction* (represented by the purple line in Fig. 2). For instance, if CEDAR were applied to the nightly version that introduces a bug, it could be detected immediately, resulting in a zero-day bug detection latency. Currently, since we have only applied CEDAR to 20 versions, the purple time window is non-zero for some bugs for two main reasons: (1) certain bugs are introduced in a version older than the earliest version among the 20 tested, and (2) some bugs are introduced in a nightly version (days or months) before a released version that we evaluate. Specifically, we calculate the *potential bug latency reduction* for the previously unknown bugs detected by CEDAR as the time between t_0 and t_1, i.e., the duration from when the bug is introduced to the release date of the version in which CEDAR first detects the bug (the purple line in Fig. 2).

Our results reveal that the average *potential bug latency reduction* is 350.5 days for the 23 bugs. If developers had applied CEDAR to the nightly versions, the time to detection for these bugs could be 0 days, potentially reducing bug latency from 350.5 to 0 days. This represents only a *potential* bug latency reduction, as we did not apply CEDAR to all nightly versions, and considering the inherent randomness of fuzzing, there is a slight possibility that CEDAR may not detect some bugs in the versions where they are introduced.

> Continuously applying CEDAR reduces bug detection latency of the evaluated libraries by nearly a year (on average 338.6 days), and can potentially further reduce it by 350.5 days, facilitating earlier bug detection.

C. Bug lifetime

Fig. 3 and Fig. 4 show the lifetime of the bugs detected by CEDAR in PyTorch and TensorFlow respectively. Among the 83 bugs, we omit bugs that either 1) have no status change during the time window we investigate, e.g., buggy in all the versions; or 2) are known bugs. For example, we omit the TensorFlow bug with

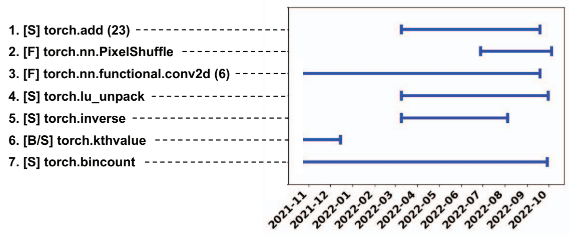


Fig. 3. PyTorch bug lifetime. Numbers in parenthesis () are the number of buggy APIs if more than one, while the y-axis lists one of them. Letters in brackets [] are the symptoms of the bugs: **S** - segmentation fault; **F** - floating point exception; **A** - abort; **B** - bus error. The left cap of a blue line denotes bug introduction time and the right cap indicates bug fixing time.

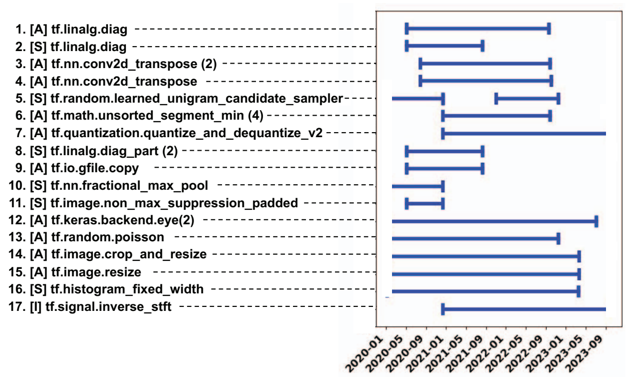


Fig. 4. TensorFlow bug lifetime. Numbers in parenthesis () are the number of buggy APIs if more than one, while the y-axis lists one of them. Letters in brackets [] are the symptoms of the bugs: **S** - segmentation fault; **F** - floating point exception; **A** - abort; **I** - inconsistency. The left cap of a blue line denotes bug introduction time and the right cap indicates bug fixing time.

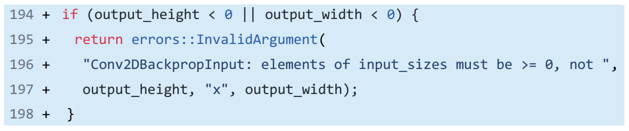
API `tf.image.combined_non_max_suppression` in Fig. 4 because the bug exists in all versions we investigated and remains unfixed until now. In total, we omit 1 and 58 bugs from PyTorch and TensorFlow respectively.

We manually investigate the bugs that CEDAR detected (Table I) on the versions listed in Table II and Table III. In the figures, the x-axes are the dates, and the y-axes represent each bug. We use the buggy API to represent each bug. Numbers in parenthesis () are the number of buggy APIs if more than one, while the y-axis lists one of them. Letters in brackets [] are the symptoms of the bug: **S** - segmentation fault; **F** - floating point exception; **A** - abort; **B** - bus error; **I** - inconsistency. For example, Bug 1 in Fig. 3 represents a PyTorch bug that causes segmentation fault ([S]) in 23 APIs including `torch.add`. [B/S] indicates that PyTorch Bug 6 in Fig. 3 results in both a bus error and a segmentation fault in the PyTorch API `torch.kthvalue`.

The blue lines represent the time windows that each bug remains open/unfixed. The time points are rounded to versions. The left cap on a line, if any, represents the time point the bug is introduced. No left cap means the bug is introduced prior to the oldest versions we investigate, i.e., PyTorch v1.10.0 and TensorFlow v2.1.0. We exclude older versions because they

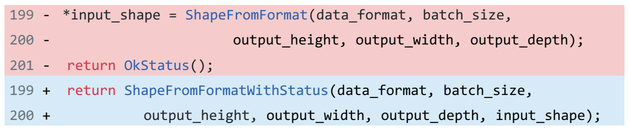
```
tf.random.learned_unigram_candidate_sampler(
  true_classes = np.array([[1000000]]),
  num_true = 1, num_sampled = 1, unique = False, range_max = 1 )
```

Fig. 5. Bug-triggering input for TensorFlow Bug 5.



```
+ if (output_height < 0 || output_width < 0) {
+   return errors::InvalidArgument(
+     "Conv2DBackpropInput: elements of input_sizes must be >= 0, not ",
+     output_height, "x", output_width);
+ }
```

a. Bug fix for TensorFlow Bug 3.



```
- *input_shape = ShapeFromFormat(data_format, batch_size,
-                      output_height, output_width, output_depth);
- return OkStatus();
+ return ShapeFromFormatWithStatus(data_format, batch_size,
+           output_height, output_width, output_depth, input_shape);
```

b. Bug fix for TensorFlow Bug 4.

Fig. 6. TensorFlow bug fixes for Bug 3 and Bug 4 for API `tf.nn.conv2d_transpose`.

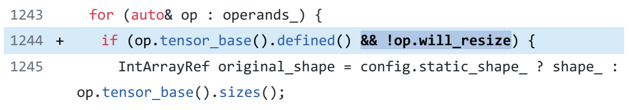
are not actively maintained and developers would not fix bugs detected in those older versions, according to a response we got from a developer. The right cap on a line, if any, represents the time point the bug is fixed. No right cap means the bug remains unfixed until now (Nov. 1st, 2023).

a) Regression Bugs: CEDAR detects new regression bugs, which is an ideal use case for CEDAR. For example, TensorFlow Bug 5 in Fig. 4 results in a segmentation fault in API `tf.random.learned_unigram_candidate_sampler` due to large values in the parameter `true_classes` as shown in Fig. 5. The bug existed in earlier versions of TensorFlow and was fixed in v2.4.0 where the API outputs the correct results. However, the bug was re-introduced in v2.7.0 released on November 4th, 2021, which has the same symptom as Bug 5 given the same bug-triggering input. We reported this new regression bug and it was fixed in Nov. 2022.

b) Masked Bugs: Furthermore, by continuously applying CEDAR on new releases, it detects additional new bugs that may be masked by existing bugs. For example, CEDAR detects two previously unknown TensorFlow bugs (Bug 3 and Bug 4 in Fig. 4) affecting the same API (`tf.nn.conv2d_transpose`) and causing similar crashes, i.e., abort. Bug 3 is triggered by negative values in the parameter `output_shape` (e.g., `output_shape=[2,-2]`), and Bug 4 is caused by a large value in the same parameter (e.g., `output_shape=[114078056, 179835296]`). After reporting Bug 3 to the TensorFlow developers, they fixed it on Sep. 26, 2022 (Fig.6a). However, the API `tf.nn.conv2d_transpose` continued to crash in the nightly version on Sep. 27, 2022, due to Bug 4. This bug was later confirmed and fixed (Fig.6b) two days after being reported. These two bugs, fixed in different locations, demonstrate the importance of continuous testing even after resolving bugs, as additional issues may be concealed. Bug 3 was addressed with a negative value check, while Bug 4 was resolved by

```
torch.add( input = torch.ones([2,2]), other = torch.ones([1]),
           out = torch.ones([2,2,1,1]) )
```

a. Bug-triggering input for PyTorch Bug 1.



```
  for (auto& op : operands_) {
+   if (op.tensor_base().defined() && !op.will_resize) {
      IntArrayRef original_shape = config.static_shape_ ? shape_ :
    op.tensor_base().sizes();
```

b. Bug fix for PyTorch Bug 1.

Fig. 7. PyTorch Bug 1 in `torch.add` and 22 additional APIs.

replacing the API `ShapeFromFormat` with a safer API `ShapeFromFormatWithStatus` to prevent crashes (e.g., due to overflow). This example highlights the importance of continuous testing in uncovering hidden (masked) bugs.

> CEDAR demonstrates its effectiveness in regression and continuous testing by detecting new *regression bugs* and *masked bugs*, both of which are challenging to detect when testing a single version.

CEDAR detects previously unknown inconsistency bugs with C-EAGLE. For instance, Bug 17 in Fig. 4 is a new inconsistency bug introduced in TensorFlow v2.4.0 (Dec. 14, 2020). The developers have confirmed this bug after we reported it. Details of Bug 17 are in Section V-D.

D. Bug examples

a) Case study 1 (PyTorch Bug 1): C-DocTer detects a segmentation fault (i.e., Bug 1 in Fig.3) in the newly added code of PyTorch v1.11.0, which causes crashes in 23 PyTorch APIs, including `torch.add` and `torch.eq`. These APIs are mathematical operations and require three input parameters: two operand parameters (`input` and `other`) and one output tensor (`out`) to store the results. The APIs crash with a segmentation fault when there is a shape mismatch, specifically when `out` has at least two more dimensions than both operands. For example, in Fig.7a, the two operands have 2 and 1 dimensions, respectively, while `out` has 4 dimensions. After we reported the bug, it was labeled as **high priority** by the developers and subsequently fixed with the patch shown in Fig.7b. This case study illustrates CEDAR's effectiveness in detecting bugs in newly added code, reducing the bug detection latency (i.e., red dashed line in Fig.2) to 0 days.

b) Case study 2 (TensorFlow Bug 16): C-DocTer detects a previously unknown TensorFlow bug in the API `tf.histogram_fixed_width` when passing large values (e.g., 3e+38) to parameters `values` and `value_range`. This bug has existed since v2.1.0. However, DocTer [7] failed to detect it due to the randomness of the fuzzing process. C-DocTer triggers this bug in two out of ten runs when testing versions v2.10.0 and v2.11.0.dev20220916. Due to the random nature of fuzz testing, some bugs can be difficult to trigger with a limited number of trials. The developers confirmed and fixed the bug after we reported it. This case study highlights

```
layer = torch.nn.PixelShuffle(1)
input = torch.ones((1,1,1,1,0))
out = layer(input)
```

a. Bug-triggering input for PyTorch Bug 2.

```
+  if (output.numel() == 0) {
+    return output;
+  }
```

b. Bug fix for PyTorch Bug 2.

Fig. 8. PyTorch Bug 2 in `torch.nn.PixelShuffle`.

```
   if (num_diags == 1) {  // Output has rank `rank+1`.
     output_shape.set_dim(diag_rank - 1, num_rows);
-    output_shape.AddDim(num_cols);
+    OP_REQUIRES_OK(context, output_shape.AddDimWithStatus(num_cols));
```

a. Bug fix for TensorFlow Bug 1.

```
+    OP_REQUIRES(context, diag_index.NumElements() > 0,
+           errors::InvalidArgument(
+               "Expected diag_index to have at least 1 element"));
```

b. Bug fix for TensorFlow Bug 2.

Fig. 9. TensorFlow bug fixes for Bug 1 and Bug 2 for API `tf.linalg.diag`.

the advantage of applying continuous testing to repeatedly test code, increasing the likelihood of exposing *hard-to-detect* bugs.

c) Case study 3 (PyTorch Bug 2): C-DocTer detects a floating-point exception in the PyTorch layer API `torch.nn.PixelShuffle` (i.e., Bug 2 in Fig.3) when providing the layer API with an empty tensor, such as the `input` in Fig.8a. We reported this bug to the PyTorch developers on GitHub, and it was labeled by the developers as **high priority**. The bug was later fixed with the patch in Fig 8b with a zero check.

d) Case study 4 (TensorFlow Bug 1 and Bug 2): C-DocTer detects two bugs (Bug 1 and Bug 2 in Fig.4) in the same TensorFlow API, `tf.linalg.diag`, but with different symptoms-—one with an abort (Bug 1) and another with a segmentation fault (Bug 2). The API returns a batched diagonal tensor with given batched diagonal values. The parameter `k` represents the diagonal offset(s), where positive values mean super-diagonal and negative values mean sub-diagonal. Bug 1 is a previously unknown bug discovered by CEDAR, caused by a large value of `k`, e.g., `k=1070828000000`. The developers fixed it after we reported it by replacing the function `AddDim` with `AddDimWithStatus`, which prevents check crashes (Fig9a). Bug 2 is a known bug introduced in v2.2.0 and fixed before v2.6.0. It is caused by an empty `k`, e.g., `k=[]`, and was fixed by adding a size check in Fig. 9b.

e) Case study 5 (TensorFlow Bug 17): C-EAGLE detects a bug by observing the inconsistency between the outputs of the eager mode and graph mode of the API `tf.signal.inverse_stft`, which are supposed to yield the same results. In the eager mode, TensorFlow evaluates operations instantly, while in the graph mode, it constructs a computational graph before evaluation, enabling compiler-level

TABLE IV
EXECUTION TIME. THE TIME IS OF FORMAT HH:MM.

Time (Avg.)	Total	C-DocTer	C-EAGLE
PyTorch	07:50	03:30	04:20
TensorFlow	09:08	04:10	04:58
Average	08:29	03:50	04:39

TABLE V
EFFECTIVENESS OF THE OPTIMIZATION STRATEGIES. THE TIME IS OF FORMAT HH:MM.

Time (Avg.)	Total	C-DocTer	C-EAGLE
CEDAR	08:29	03:50	04:39
CEDAR - Parallelism	95:53	34:43	61:10
CEDAR - Test case reduction	11:42	03:50	07:52
CEDAR - Parallelism - Test case reduction	130:36	34:43	95:53

optimizations. Both modes are expected to produce equivalent outputs. The developers have confirmed this bug after we reported it.

f) Case study 6 (PyTorch `remainder` Bug): C-EAGLE detects inconsistencies when passing inputs with different data types to the API `torch.remainder`. Specifically, C-EAGLE casts the input parameter into two data types, $type_X$, and $type_Y$, and feeds them into the API. If the input value is within the intersection of two data types' size ranges, the outputs are expected to be the same. However, we observed a significant difference among the outputs, as large as 197, when passing the same value in different data types (`torch.int32` and `torch.float32`). After reporting this inconsistency, the developers confirmed it as a numerical stability problem.

E. Execution time

In Table IV, we list the execution time (in the format of HH:MM) of CEDAR (Col. *"Total"*) on PyTorch and TensorFlow respectively, as well as the time breakdown for C-DocTer and C-EAGLE. On average, CEDAR takes 08:29 hours to conduct testing on PyTorch or TensorFlow. On average, it takes 0.14 seconds for C-DocTer to generate and test each input, and 24.9 minutes for C-EAGLE to execute a pair of equivalent graphs.

F. Optimization

In this section, we study the effectiveness of the optimization strategies (Section III-E). We focus on the time efficiency for strategies *parallelism* and *test case reduction*, and space efficiency for strategy *redundancy removal*. We also discuss the bug detection effectiveness for *test case reduction*.

Table V shows the test time reduction and the ablation study of our optimization strategies *parallelism* and *test case reduction*. The table lists the average execution time (HH:MM) of PyTorch and TensorFlow for CEDAR, CEDAR without *parallelism*, CEDAR without *test case reduction*, and CEDAR without *parallelism* and *test case reduction*. In each row, we list the execution time for C-DocTer, C-EAGLE, and the total time (i.e., time for CEDAR in Col. *"Total"*). The results show that the two optimization strategies combined effectively reduce the testing time overhead by a factor of 15.4. Specifically, the *test case reduction* strategy reduces the continuous testing time overhead by a factor of 1.4, while *parallelism* reduces it by a factor of 11.3.

Although the *test case reduction* strategy only helps reduce the execution time of 3 hours and 13 minutes on average, it increases the chance of triggering bugs or inconsistencies. According to a previous study [7], only 33.4% of the inputs that DocTer generates are likely valid (i.e., do not cause the APIs to crash or throw exceptions). Invalid inputs would be rejected by the function's input validity check and are not likely to trigger any inconsistencies. Exercising invalid inputs would not only increase time overhead but also lower the chance of triggering a bug with limited time and resources. With the *test case reduction* strategy, C-DocTer only feeds C-EAGLE with `E_max_iter` likely valid inputs, and it, therefore, increases the chance of triggering inconsistencies.

Our experiments also show that the strategy *redundancy removal* helps remove 1,492,538 and 1,648,391 redundant files and releases 64.25GB and 94.97GB space for PyTorch and TensorFlow respectively. Since CEDAR only uses on average 8.2G storage for all the logs, outputs, and bug-relevant inputs, the strategy reduces the space overhead by a factor of 9.7.

> The optimization strategies, which combine generic and tool-specific approaches, effectively reduce the time and space overhead by factors of 15.4 and 9.7, making CEDAR's continuous testing more practical.

VI. Threats to validity

One threat is that older versions of documents may contain outdated constraints, which could lead to false positive bugs. We did not observe false positives due to this reason in our experiments. This is because the main negative impact of using outdated documents is increasing the invalid input ratio, i.e., the ratio of generated inputs that trigger exceptions, which would not introduce false positive bugs for C-DocTer and C-EAGLE, as they only focus on crashes and inconsistencies. In the future, one can mitigate this threat by analyzing the latest versions of the documents.

Moreover, since we only investigate official and certain nightly versions of each library within a certain time window (Section IV) and the time points are rounded to versions in Section V-B, the bug introduction time (Fig. 2) may not be precise, which puts a threat to the validity of the potential bug latency reduction (350.5 days). However, the bug introduction time could only be earlier than the time that we obtain, as a bug could be introduced in a nightly version before the bug-introducing version that we identified. Therefore, CEDAR could potentially reduce the bug latency by 350.5 days minimally.

VII. Related Work

a) Continuous Testing: Continuous testing [17] is testing in a situation where software is continuously developing and can be released at any time. With the growth of modern software projects' code sizes, there's an increasing reliance on continuous integration systems [27]. There has been significant research on optimizing this procedure [15], [16], [18], [28]. Google researchers [16] proposed test prioritization to reduce workload by avoiding frequent re-execution of unlikely-to-fail tests. CEDAR also employs optimization strategies such as test case reduction (Section III-E). While previous works focus on test case selection, prioritization, etc., our approach targets DL libraries, integrating the state-of-the-art DL testing frameworks, with tool-specific optimizations.

b) Bug life cycle: There has been work studying the bug fixing time, i.e., from bug reported till bug report closed, for bugs in open-source projects and found that it takes less than three months for most bugs to be fixed in large open-source projects (e.g., Eclipse) [29], and 0-1.5 months for most bugs in open-source Android apps [30].

c) Bugs in DL/ML systems: There is research on the bugs in the DL or ML systems [31], [32], including their categories (e.g., data bug), root causes (e.g., unaligned tensors), challenges for debugging (e.g., probabilistic correctness), etc. We focus on crashes and inconsistencies and provide a study on the life cycles of the bugs detected, including the bug-fixing time, bug detection latency, etc.

d) Testing DL libraries: There are approaches focused on testing DL libraries [8], [9], [11], [12]. Fuzz-based approaches leverage information extracted from documents [7], [9]–[11], code base [8], [10], and gradients [12]. Different from them, CRADLE [5], LEMON [33], EAGLE [6], DeepREL [9], and ▽Fuzz [11] resolves the oracle challenge with differential testing. CEDAR selects two latest DL testing approaches to study their integration in a continuous testing setup.

VIII. Conclusion and Discussion

In this paper, we explore DL testing approaches in a continuous testing setting, examining their capabilities and effectiveness in finding bugs in continuous builds. We propose a continuous testing framework for DL libraries, CEDAR, which efficiently integrates two state-of-the-art DL testing approaches, i.e., DocTer and EAGLE, to test DL libraries for detecting crashes and inconsistency bugs. We share our conclusions, insights, and lessons learned for the reference of future developers and researchers.

a) Effectiveness in continuous testing: One insight gained from this integration is that continuous testing of cutting-edge tools increases test effectiveness. Specifically, CEDAR is effective in finding bugs from the *newly added code* within both nightly and released versions, as well as hard-to-detect bugs within the *existing code* from DL libraries. In addition, CEDAR identifies *regression bugs* and *masked bugs*, both of which are challenging to detect when testing a single version. Overall, both C-DocTer and C-EAGLE effectively detect bugs in the tested versions. By continuously applying CEDAR, it detects 83 bugs from 140 APIs in 20 versions of PyTorch and TensorFlow. Out of the 83 bugs, 23 are previously unknown bugs, with 21 confirmed or fixed by the developers.

b) Efficient integration with tool-specific optimizations: Another lesson learned from our integration with a leading company is that speed is crucial for the practical adoption of CI testing tools. One best practice is combining generic and tool-specific optimizations. Specifically, to make the integration more efficient, we design and implement three optimization strategies that reduce CEDAR's time and space overhead by a factor of 15.4 and 9.7. While one of the optimization strategies (*parallelism*) is general, *test case reduction* and *redundancy removal* are tailored specifically for the integration of DocTer and EAGLE. To make the integration of domain-specific tools more efficient, specialized (joint) optimizations that consider the unique characteristics of each tool are needed.

c) Reducing bug detection latency: Our results (Section V-B) show that CEDAR has effectively shortened the bug detection latency by almost a year (338.6 days) on average. If applied to all nightly versions, CEDAR could further reduce bug detection latency substantially. This underscores the importance of continuous testing on DL libraries and highlights the potential of integrating domain-specific tools in the testing process.

We explore integrating state-of-the-art DL library testing techniques with continuous testing, making them more efficient and effective for both developers and researchers. Our work demonstrates the importance of domain-specific optimizations and the potential impact of continuous testing on detecting bugs and reducing bug detection latency in the development of DL libraries.

Acknowledgment

The authors thank the anonymous reviewers for their invaluable feedback. This work is partially supported by NSF 1901242, NSF 2006688, and Meta/Facebook Research Awards.

References

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 2722–2730.

[2] M. Popel and J. e. a. Tomkova, M.and Tomek, "Transforming machine translation: a deep learning system reaches news translation quality comparable to human professionals," 2020.

[3] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[4] "American fuzzy lop," 2013. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[5] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1027–1038.

[6] J. Wang, T. Lutellier, S. Qian, H. V. Pham, and L. Tan, "EAGLE: Creating equivalent graphs to test deep learning libraries," 2022.

[7] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "DocTer: documentation-guided fuzzing for testing deep learning api functions," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 176–188.

[8] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "IvySyn: Automated vulnerability discovery in deep learning frameworks," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2383–2400. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/christou

[9] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Fuzzing deep-learning libraries via large language models," *arXiv preprint arXiv:2212.14834*, 2022.

[10] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 995–1007.

[11] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," *arXiv preprint arXiv:2302.04351*, 2023.

[12] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 530–543.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for Large-Scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[15] E. Fallahzadeh and P. C. Rigby, "The impact of flaky tests on historical test prioritization on chrome," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 273–282.

[16] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 233–242.

[17] D. Saff and M. Ernst, "Reducing wasted development time via continuous testing," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 281–292.

[18] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[19] "Cedar," 2023. [Online]. Available: https://github.com/lt-asset/cedar

[20] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE/ACM, July 2020.

[21] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 510–520.

[22] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 2020 International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, 07 2018, pp. 129–140.

[23] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020.

[24] "DocTer," 2022. [Online]. Available: https://github.com/lin-tan/DocTer

[25] "EAGLE," 2022. [Online]. Available: https://github.com/lin-tan/eagle

[26] "cron," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Cron

[27] K. Gallaba, M. Lamothe, and S. McIntosh, "Lessons from eight years of operational data from a continuous integration service: an exploratory case study of circleci," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1330–1342.

[28] D. Elsner, R. Wuersching, M. Schnappinger, A. Pretschner, M. Graber, R. Dammer, and S. Reimer, "Build system aware multi-language regression test selection in continuous integration," in *Proceedings of*

the 44th International Conference on Software Engineering: Software Engineering in Practice, 2022, pp. 87–96.

[29] L. Marks, Y. Zou, and A. E. Hassan, “Studying the fix-time for bugs in large open source projects,” in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, 2011, pp. 1–8.

[30] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, “An empirical analysis of bug reports and bug fixing in open source android apps,” in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 133–143.

[31] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[32] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, “An empirical study on tensorflow program bugs,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[33] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, “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*, 2020, pp. 788–799.