History-Driven Fuzzing For Deep Learning Libraries

NIMA SHIRI HARZEVILI, York University, Canada
MOHAMMAD MAHDI MOHAJER, York University, Canada
MOSHI WEI, York University, Canada
HUNG VIET PHAM, York University, Canada
SONG WANG, York University, Canada

Recently, many Deep Learning (DL) fuzzers have been proposed for API-level testing of DL libraries. However, they either perform unguided input generation (e.g., not considering the relationship between API arguments when generating inputs) or only support a limited set of corner-case test inputs. Furthermore, many developer APIs crucial for library development remain untested, as they are typically not well documented and lack clear usage guidelines, unlike end-user APIs. This makes them a more challenging target for automated testing.

To fill this gap, we propose a novel fuzzer named Orion, which combines guided test input generation and corner-case test input generation based on a set of fuzzing heuristic rules constructed from historical data known to trigger critical issues in the underlying implementation of DL APIs. To extract the fuzzing heuristic rules, we first conduct an empirical study on the root cause analysis of 376 vulnerabilities in two of the most popular DL libraries, PyTorch and TensorFlow. We then construct the fuzzing heuristic rules based on the root causes of the extracted historical vulnerabilities. Using these fuzzing heuristic rules, Orion generates corner-case test inputs for API-level fuzzing. In addition, we extend the seed collection of existing studies to include test inputs for developer APIs.

Our evaluation shows that Orion reports 135 vulnerabilities in the latest releases of TensorFlow and PyTorch, 76 of which were confirmed by the library developers. Among the 76 confirmed vulnerabilities, 69 were previously unknown, and 7 have already been fixed. The rest are awaiting further confirmation. For end-user APIs, Orion detected 45.58% and 90% more vulnerabilities in TensorFlow and PyTorch, respectively, compared to the state-of-the-art conventional fuzzer, DeepRel. When compared to the state-of-the-art LLM-based DL fuzzer, AtlasFuz, and Orion detected 13.63% more vulnerabilities in TensorFlow and 18.42% more vulnerabilities in PyTorch. Regarding developer APIs, Orion stands out by detecting 117% more vulnerabilities in TensorFlow and 100% more vulnerabilities in PyTorch compared to the most relevant fuzzer designed for developer APIs, such as FreeFuzz.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Fuzz testing, test generation, deep learning

ACM Reference Format:
Nima Shiri Harzevili, Mohammad Mahdi Mohajer, Moshi Wei, Hung Viet Pham, and Song Wang. 2018. History-Driven Fuzzing For Deep Learning Libraries. 1, 1 (July 2018), 29 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' addresses: Nima Shiri Harzevili, nshiri@yorku.ca, York University, 4700 Keele St., North York, Ontario, Canada, M3J 1P3; Mohammad Mahdi Mohajer, mmm98@yorku.ca, York University, 4700 Keele St., North York, Ontario, Canada, M3J 1P3; Moshi Wei, moshiwei@yorku.ca, York University, 4700 Keele St., North York, Ontario, Canada, M3J 1P3; Hung Viet Pham, hvpham@yorku.ca, York University, 4700 Keele St., North York, Ontario, Canada, M3J 1P3; Song Wang, wangsong@yorku.ca, York University, 4700 Keele St., North York, Ontario, Canada, M3J 1P3.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
Manuscript submitted to ACM

1 INTRODUCTION

The use of deep learning (DL) in modern software systems is now widespread and continues to grow. Many successful safety-critical applications have been built on top of DL libraries such as autonomous driving systems [25, 63, 65], healthcare [6, 61], and financial services [22]. As a consequence, there has been a growing concern about vulnerabilities in DL systems. Moreover, as DL applications become new targets for malicious attacks [29, 33], reliability and robustness become critical requirements for these applications due to their potential impact on human life [4, 8, 9, 11, 16, 30, 39, 47].

In recent years, many fuzzers have been proposed for fuzzing DL APIs [19–21, 53, 55]. Specifically, FreeFuzz [53] performed API-level testing via random fuzzing on end-user APIs of TensorFlow and PyTorch collected from three different sources, i.e. API reference documentation, DL models in the wild, and developer tests. DeepRel [21] extended FreeFuzz by random mutation of semantically related APIs in terms of equivalence values and statuses. DocTer [55] performed fuzz testing via automatic extraction of constraints from the API reference documentation of three DL libraries (i.e., TensorFlow, PyTorch, and MXNet). TitanFuzz [19] leveraged the power of Large Language Models (LLMs) [23, 54, 60] to generate inputs tailored for testing DL APIs. AtlasFuzz [20] extended TitanFuzz by including unusual programs mined from open source by LLMs to guide the generation of test inputs. Although existing DL fuzzers have shown great potential in finding real-world DL vulnerabilities, they suffer from the following limitations:

Challenge 1: Unguided Input Generation. Existing DL fuzzers [19–21, 53, 55] mainly treat the parameters of DL APIs independently in API-level fuzz testing. However, during our empirical study, we find that some vulnerabilities require a special combination of different input arguments, e.g., `mismatch between input arguments` of DL APIs is one of the major root causes of reported security vulnerabilities. Figure 1 shows an example vulnerability[1] exposed by *mismatch between dimensions of the input tensors* in earlier releases of TensorFlow. The description of the vulnerability report states that the implementation of *tf.raw_ops.SparseTensorDenseAdd* lacks validation of the input arguments, resulting in undefined behavior. According to the API reference documentation of *tf.raw_ops.SparseTensorDenseAdd*, the first dimension of *a_values* is expected to match with the first dimension of *a_indices*, and the first dimension of *a_shape* should align with the second dimension of *a_indices*. This complicated relationship imposes a challenge for existing fuzzers, as they cannot typically explore and discover such nuanced correlations among the input arguments of DL APIs. As a result, triggering vulnerabilities exposed by *guided input generation* becomes practically implausible within the scope of conventional and LLM-based fuzzing techniques.

Solution: Guided Input Generation using Historical Rules. To address **Challenge 1**, we propose guiding the generation of test inputs for fuzzing DL APIs using heuristic rules derived from historical vulnerabilities triggered by specific DL API argument patterns. *Our key insight on the guided input generation of test inputs is that security vulnerabilities in DL libraries often manifest when APIs encounter a mismatch between input arguments.* Figure 1 (the lower box) provides an example fuzzing heuristic rule and the corresponding generated test case illustrating our approach to solving the aforementioned problem by focusing on the `mismatch between input arguments`. Initially, we perform a root cause analysis by summarizing a set of historical vulnerabilities attributed to `mismatches between input arguments from DL APIs` (see Section 2). Using the insights from this root cause analysis, we formulate a set of fuzzing heuristics rules designed to produce targeted test inputs capable of exposing such vulnerabilities (refer to Table 3 for a detailed explanation of these fuzzing heuristics rules). As shown in Figure 1, the constructed rule, in the form of a mutator function, alters the dimensions of the input tensors, ultimately triggering a segmentation fault. More specifically, the generated test case features two input tensors with conflicting dimensions: the first tensor possesses a

[1] https://github.com/tensorflow/tensorflow/security/advisories/GHSA-rc9w-5c64-9vqq

(3D) shape, while the second tensor has a (2D) shape. The generated test inputs violate the constraints outlined in the API reference documentation for *tf.raw_ops.lower_bound*, which specifies that both *sorted_inputs* and *values* should be (2D) tensors.

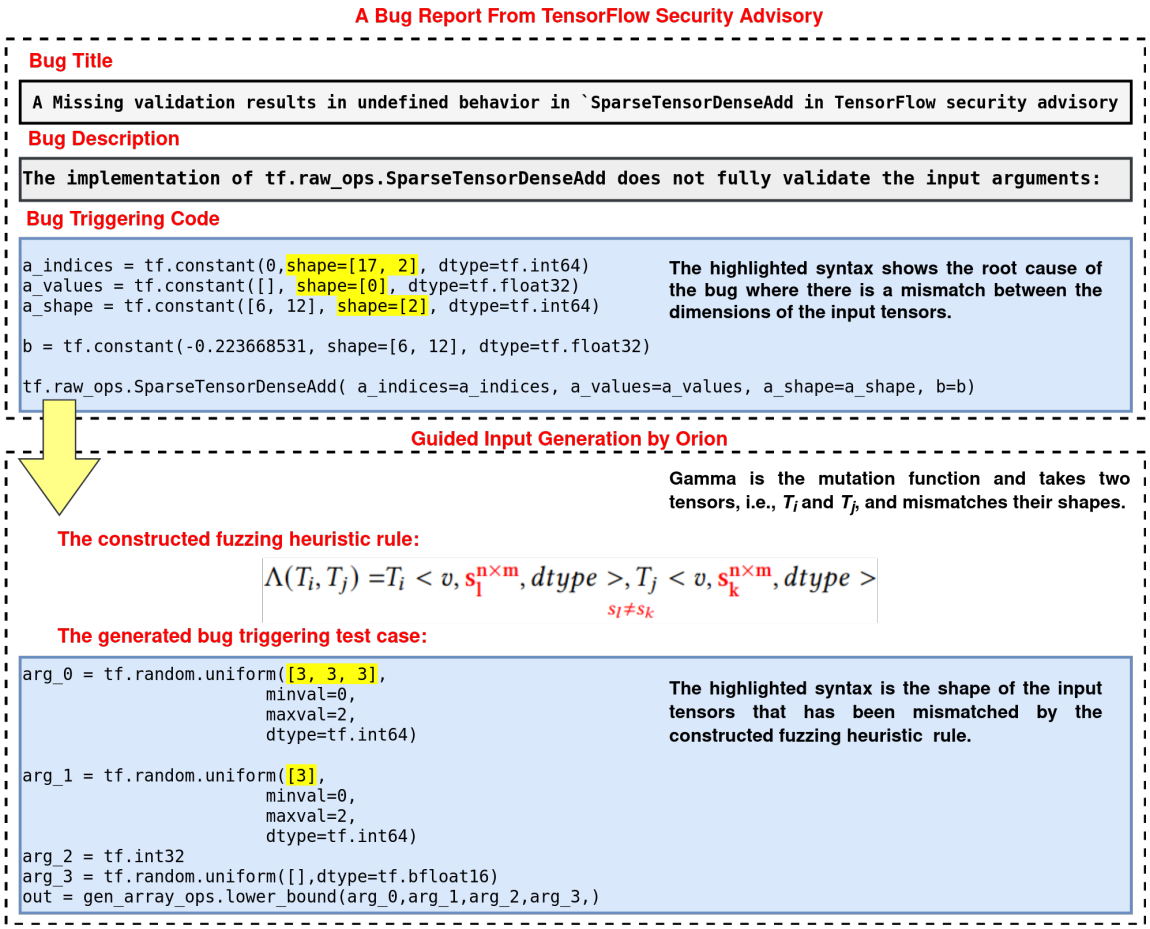


Fig. 1. A motivational example that shows how Orion used TensorFlow security advisory to build a fuzzing heuristic rule that models the correlation between input parameters of the API *gen_array_ops.lower_bound* which eventually resulted in the detection of a segmentation fault.

Challenge 2: Lack of Support for Corner Case Generation. Existing conventional DL fuzzers at the API level [19–21, 53, 55] suffer from two primary limitations when it comes to generating corner case test inputs. First, their heuristics rules for corner case test input generation are *random* and *unguided*. Random values may not accurately replicate real-world vulnerabilities or tainted inputs provided by external attackers. API calls often rely on specific data formats, constraints, or patterns that random values may not adhere to. This can lead to missing the detection of critical vulnerabilities. For example, consider a documented issue within the TensorFlow GitHub repository[2]. In this case, a

[2]https://github.com/tensorflow/tensorflow/issues/51908

TensorFlow user reported a crash in the *tf.pad* API when called with `large padding sizes`. Our proposed fuzzer, `Orion`, successfully identified this security vulnerability, while it went unnoticed by other studied DL fuzzers. The reason behind this lies in their approach of generating arbitrary test inputs for DL APIs, rendering them essentially incapable of detecting vulnerabilities that manifest only under extreme corner-case conditions. As for LLM-based DL fuzzers like TitanFuzz and AtlasFuzz, they primarily focus on addressing logical bugs by modeling context information and DL API usage call sequences, rather than detecting security vulnerabilities through corner case test input generation.

Solution: Guided Corner Case Generation using Historical Rules. To effectively address the challenge of limited corner case test input generation, we analyze historical security vulnerability data to identify fuzzing heuristic rules about the types of input, corner case conditions, or unusual scenarios that have historically resulted in software security vulnerabilities. These corner case fuzzing heuristic rules (as presented in Table 3) serve as guidance for `Orion`'s corner case generator, instructing it on how to intelligently generate or mutate test inputs. *What makes these constructed fuzzing heuristic rules particularly valuable, compared to the traditional and LLM-based DL fuzzing techniques, is their ability to accurately mimic real-world corner cases that have historically exposed software vulnerabilities.*

Challenge 3: Lack of Testing Developer APIs. Current conventional and LLM-based DL fuzzers [19–21, 53, 55] predominantly concentrate their testing efforts on end-user DL APIs. This focus arises from the fact that DL libraries often lack comprehensive documentation and usage guidelines for developer APIs, in contrast to the well-documented nature of end-user APIs.

Solution: Modeling Developer API Context Information. To overcome the challenge, we developed a lightweight static analyzer and collected developer API context information heuristically. The core idea is to focus on internal Python modules within the DL libraries where all developer APIs are located which act as an intermediate between Python client APIs, i.e., end-user APIs, and the backend implementation of DL libraries.

Our assessment demonstrates that `Orion` has identified a total of 135 vulnerabilities on the latest releases of TensorFlow (2.13.0) and PyTorch (1.13.1), and 76 of them have been verified by library developers. Of these 76 confirmed vulnerabilities, 69 were previously unknown and 7 have already been addressed and resolved. The remaining bugs are pending further confirmation. Specifically, **Regarding end-user APIs**, `Orion` uncovers 45.58% and 90% additional vulnerabilities in TensorFlow and PyTorch compared to the leading conventional DL fuzzer, that is, DeepRel. Compared to the cutting-edge LLM-based fuzzer, i.e. AtlasFuzz, `Orion` could detect 13.63% and 18. 42% more vulnerabilities in TensorFlow and PyTorch, respectively. **In terms of developer APIs**, `Orion` could outperform FreeFuzz (the only applicable baseline) by detecting 117% and 100% more vulnerabilities on TensorFlow and PyTorch respectively. This paper makes the following contributions:

- We present a simple yet effective approach to API-level testing of deep learning libraries. To the best of our knowledge, this is the first study to leverage historical vulnerability data for the creation of fuzzing heuristic rules.
- We characterize and categorize a set of fuzzing heuristic rules based on an empirical study of 376 security records extracted from the TensorFlow security advisory[3] and GitHub repository of PyTorch.
- We design and develop a DL fuzzer, `Orion`, that utilizes the fuzzing heuristics rules to guide its input generation. `Orion` also instruments both end-user and developer APIs specifically designed to test downstream components of DL libraries.

[3]https://github.com/tensorflow/tensorflow/security/advisories.

- `Orion` can find 135 security vulnerabilities of which 76 of them are already confirmed by library developers. Among the confirmed security vulnerabilities, 69 of them are new, and 7 of them are already fixed. The rest of the reported security vulnerabilities are pending and awaiting further confirmation.

2 EMPIRICAL ANALYSIS OF HISTORY SECURITY VULNERABILITIES

To characterize and gain insight into fuzzing heuristic rules, we conducted an empirical study on 376 historical security vulnerabilities in PyTorch and TensorFlow to understand their root causes. In this section, we explain our approaches to collecting and analyzing these records, as well as extracting the fuzzing heuristic rules that `Orion` employs to guide its fuzzing operations.

2.1 Data Collection

In this work, we gathered historical security vulnerability data from two different sources for TensorFlow and PyTorch: issues in the GitHub repository for PyTorch and the TensorFlow security advisory[4]. The reason for collecting security records from GitHub for PyTorch is that, at the time of writing this paper, there were only four reported security vulnerabilities in its CVE repository[5]. For TensorFlow, we meticulously reviewed all 407 reports available at the time of writing this paper on its security advisory page.

To prevent data leakage and ensure the fuzzing heuristic rules maintain their generalizability, we adopt a strategy of using earlier releases of PyTorch and TensorFlow to craft the rules, while employing the most recent releases of each library as test data. For TensorFlow, we leverage versions 2.3.0 through 2.10.0 as historical data, with release 2.13.0 serving as the test data. Similarly, for PyTorch, we utilize versions 0.4.1 to 1.13.0 as historical data, while employing release 1.13.1 as the test dataset. This approach helps to ensure that our fuzzing heuristic rules remain robust and applicable across different versions while also validating their effectiveness on the latest releases of the libraries.

2.1.1 Automated collection of real-world security vulnerabilities. For the automatic collection of issues from PyTorch, we iterated through all issues available in its repository. We used keyword-based matching approaches, similar to those used in [49, 51, 66], to automatically filter irrelevant issues and collect those related to security vulnerabilities. Note that filtering rules are only applied for PyTorch. For TensorFlow, we only collected records from its security advisory and since those records have been assigned CVE IDs, we did not perform any automatic and manual filtering for TensorFlow.

The chosen keywords for the filtering include: **Numerical and Memory-related vulnerabilities**: *buffer overflow*, *integer overflow*, *integer underflow*, *heap buffer overflow*, *stack overflow*, and *null pointer dereference*[6]. **Logical vulnerabilities**: *wrong result*, *unexpected output*, *incorrect calculation*, *inconsistent behavior*, *unexpected behavior*, *incorrect logic*, *wrong calculation*. **Performance vulnerabilities**: *slow*, *high CPU usage*, *high memory usage*, *poor performance*, *slow response time*, *performance bottleneck*, *performance optimization*, *resource usage*, *race condition*, *memory leak*. As a result, we collected 1,739 vulnerability-related issues from the PyTorch GitHub repository. However, our manual analysis revealed that such automated approaches produce numerous false positives. This occurs because not all issues in the PyTorch GitHub repository pertain to actual software vulnerabilities. Many unrelated issues involve the CI infrastructure, documentation, and feature requests. In the following section, we will discuss our manual filtering approach to remove these unrelated issues.

[4]https://github.com/tensorflow/tensorflow/security/advisories
[5]https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=pytorch
[6]To save space, we have not incorporated all security-related terms in the manuscript. However, a comprehensive list of all keywords can be found in the GitHub repository associated with this paper.

Table 1. Statistics of our curated dataset used for empirical study. The numbers in parenthesis are the total number of unique records for issues and CVE.

Library	# Issues	# CVE	# PR	# Commits
PyTorch	98 (89)	N.A	19	50
TensorFlow	N.A	278 (220)	8	236

2.1.2 Manual filtering. We manually examined the collected data to obtain the final list of vulnerabilities related to PyTorch. Specifically, for each issue, we carefully reviewed the issue title, description, discussions, log messages, etc. We performed the manual analysis in two rounds. Round 1: Three authors independently reviewed the PyTorch records. The authors extracted multiple pieces of information for each bug, that is, the buggy API, the impact of the bug (log messages or stack traces), how the bug can be triggered, and the bug description. Once the information is extracted, the authors cross-check the labeled bugs to mark possible disagreements. Round 2: All authors were involved in the manual analysis of the records in Round 1, and disagreements were resolved with group discussions. At the end of the round, we discarded any bug in which the authors could not reach a consensus.

Note that the following types of bugs, which are out of scope, were excluded:

- Vulnerabilities specific to certain platforms, such as Windows, Android, or iOS.
- Build and configuration issues.
- Bugs arising from external libraries such as torchvision or torchaudio.
- Bugs that do not require the input parameters to be triggered.

After applying these exclusion criteria, we identified a total of 98 issues related to security vulnerabilities in PyTorch. In total, our root cause analysis encompassed 376 DL security vulnerability records, with 278 records sourced from TensorFlow and 98 records obtained from PyTorch. Table 1 shows the total number of issue records and CVE records for PyTorch and TensorFlow (including unique records for each library), as well as the total number of unique pull requests and commits. We can also observe that the total number of commits of TensorFlow is higher than the total number of unique CVE records, the reason is that some CVE records have multiple commits. We did not use issues for TensorFlow and solely relied on CVE records that are already confirmed as security records. On the other hand, since PyTorch does not have any reported CVEs, we rely on the reported issues in their GitHub repository.

2.2 Construction of Fuzzing Heuristics Rules

To construct fuzzing heuristic rules, we initially performed manual analyses of the vulnerability records collected to explore potential input patterns that contribute to these vulnerabilities. For CVE records, we first thoroughly examined the description of reported vulnerabilities and the provided links to the TensorFlow security advisory. Subsequently, we reviewed the vulnerability description, minimum reproducing example, and the link to the commit that patched the security issue. Regarding commits, we focused on code changes to identify the root cause of the issue in the backend implementation. In the case of PyTorch security records, our review encompassed issue descriptions, discussions, related issues, and reproducing examples, all aimed at identifying the factors contributing to vulnerabilities. Specifically, we extracted the following factors from each record:

Table 2. The notations we used in this paper.

Notation	Explanation
Λ	A mutation function.
$T_j < v, s_{i,j}^{D_{1,2,...,r}}, dtype >$	A triplet which defines a tensor.
v	The value of the tensor subject to $v \in \{\mathbb{Z}, \mathbb{R}, \mathbb{C}\}$.
$s_{i,j}$	Is the dimension of the tensor in the current rank.
$s^{D1,2,...,r}$	Is the rank of tensor.
$dtype$	Denotes the tensor data types subject to $dtype \in DT$.
$i, j, r,$	A set of indices subject to $i, j, r \in \mathbb{Z} \quad \wedge \neq \mathbb{C}\|\mathbb{R}$.
$case_x$	Large/Zero value corner case generator subject to $case_x \in \mathbb{R}\|case_x \in \mathbb{Z}$.
$case_n$	Negative value corner case generator subject to $case_n \in \mathbb{R}\|case_n \in \mathbb{Z}$.
$case_{nan}$	NaN corner case generator.
$case_{none}$	Python *None* corner case generator.
$case_{mt}$	Empty value corner case generator.
$case_{noa}$	Non-ASCII character string corner case generator.
arg	A dummy argument which can take any type.
L	denotes any Python list.

Table 3. Summary of constructed fuzzing heuristic rules and their corresponding distribution for PyTorch and TensorFlow.

No	Guided Input Generation Rules	Rule Notation	PyTorch	TensorFlow
1	Tensors Shape Mismatch	$\Lambda(T_1, T_2, ..., T_j) = T_1 < v, \mathbf{s_{1_{i,j}}^{D_{1,2,...r}}}, dtype >, ..., T_j < v, \mathbf{s_{k_{i,j}}^{D_{1,2,...r}}}, dtype >$ $_{s_1^{D_{1,2,...,r}} \neq s_k^{D_{1,2,...,r}}}$	15	70
2	Tensors Dimension Mismatch	$\Lambda(T_l, arg_i) = \{T_1 < v, \mathbf{s_{i,j}^{D_{1,2,...r}}}, dtype >, \mathbf{arg_i}\}$ $_{\mathbf{arg_i} \in \mathbb{Z}}$ $_{arg_i \neq s^{D_{1,2,...,r}}}$	2	5
3	Tensors List-Indices Mismatch	$\Lambda(T_l, L_i) = T_l < v, \mathbf{s_{i,j}^{D_{1,2,...r}}}, dtype >, \mathbf{L_i} = \{k_1, k_2, ..., k_j\}$ $_{\|L_i\| \neq \|D^{1,2,...,r}}$	0	9
4	List Indices Elements Mismatch	$\Lambda(T_l, L_i) = T_l < v, \mathbf{s^{D_{1,2,...r}}}, dtype >, L_i = \{k_1, k_2, ..., \mathbf{k_j}\}$ $_{\mathbf{L_{ij}} \neq \mathbf{r}}$	0	3
5	List Indices Length Mismatch	$\Lambda(L_i, L_j) = \mathbf{L_i} = \{x_1, x_2, ..., x_n\}, \mathbf{L_j} = \{a_1, a_2, ..., a_k\}$ $_{\mathbf{\|L_i\| \neq \|L_j\|}}$	0	10
	Corner Case Input Generation Rules	**Rule Notation**		
6	Tensor Corner Case Generator Type 1	$\Lambda(T_i) = T_i < \{\mathbf{case_x}\|\mathbf{case_n}\|\mathbf{case_{mt}}\|\mathbf{case_{nan}}\}, s_{i,j}^{D_{1,2,...r}}, dtype >$	37	70
7	Tensor Corner Case Generator Type 2	$\Lambda(T_i) = T_i < v, s_{i,j}^{\mathbf{D_{case_x}, case_n, case_{mt}}}, dtype >$	6	4
9	Scalar Tensor Corner Case Generator	$\Lambda(T_i) = T_i < v, s_{i,j}^{D_{1,2,...r}}, dtype >$ $_{r=1}$	2	8
10	Non-Scalar Tensor Corner Case Generator	$\Lambda(T_i^{s=1}) = T_i < v, s_{i,j}^{D_{1,2,...r}}, dtype >$	0	22
11	Preemptive Corner Case Generator Type 1	$\Lambda($ $_{arg_i \in \mathbb{Z}\|\mathbb{R}}$ arg_i $) = case_x\|case_n\|case_{nan}\|case_{none}\|case_{mt}\|case_{noa}$	19	22
13	Preemptive Corner Case Generator Type 2	$\Lambda($ $_{\mathbf{arg_i \in String\|arg_i \in Boolean}}$ $\mathbf{arg_i}$ $) = \{\mathbf{case_x}\|\mathbf{case_n}\|\mathbf{case_{nan}}\|\mathbf{case_{none}}\|\mathbf{case_{mt}}\|\mathbf{case_{noa}}\}$	1	10
14	List Corner Case Generator	$\Lambda(L_i) =$ $_{k_j \in \{\mathbf{case_x}\|\mathbf{case_n}\|\mathbf{case_{nan}}\|\mathbf{case_{none}}\|\mathbf{case_{mt}}\|\mathbf{case_{noa}}\}}$ $L_i = \{k_1, k_2, \ldots, k_j\}$	10	37
15	Mutate Tensor Data Type	$\Lambda(T_i) = T_j < v, s_{n \times m}, dtype >$	6	8
Overall			98	278

Root cause. This factor represents the root cause of vulnerabilities in DL libraries, which is crucial when constructing fuzzing heuristic rules. We extracted 33 unique root cause categories[7] from 376 security records for both TensorFlow and PyTorch.

Reproducing example. We checked for the presence of stand-alone code examples to reproduce the vulnerability. Reproducing examples helps us understand which input specifications to DL APIs trigger vulnerabilities, aiding in the implementation of fuzzing heuristic rules.

Vulnerable Parameter and and its Type. We also extracted information about the vulnerable parameter in the API input specification and its type. Collecting the type of vulnerable parameter is important because each parameter type has its weaknesses in terms of fuzz testing. For instance, concerning tensors, two main vulnerable components are typically identified: tensor values and tensor shapes.

Ultimately, we categorized various fuzzing heuristic rules based on the 33 unique categories of root causes, and the details are presented in Table 3. It is important to note that the definition of fuzzing heuristic rules is consistent for both libraries, with the only difference being their implementation, which is specific to each library. In Table 3, we divide the fuzzing heuristic rules into two sections: *Guided Input Generation Rules* and *Corner Case Input Generation Rules*. For the rules in each category, we implemented an indicator function denoted as Λ, which acts as a mutator. The first section consists of fuzzing heuristic rules that involve mismatches in the input arguments of DL APIs. We explain the rules in the following sections:

Tensors Shape Mismatch: The mutator function, Λ, takes input tensors, T_1 to T_j, and performs a shape mismatch operation sequentially. To better understand this process, consider the manipulation of two tensors, T_1 and T_2, within the parameter space of the API under test. The sequence of operations begins by altering the shape of the first tensor, T_1. This shape modification can be achieved through either `Rank Reduction` or `Rank Expansion`, with the choice between these two operations being random. `Orion` keeps track of the previous operation in a temporary variable for reference. Now, as the shape mutation of the second parameter begins, the process retrieves the previously recorded operation from the temporary variable. Then it performs the opposite operation compared to what was executed on the first parameter. In other words, if the initial operation on the first parameter was `Shape Reduction`, the new operation applied to the second parameter is `Shape Expansion`, and vice versa. This rule ensures that the shape mismatch is consistently executed as intended.

Tensor Dimension Mismatch: The mutator function associated with this rule requires two inputs: tensor T_l and an integer argument arg_i. In this context, the integer parameter serves as an indicator for the dimension of T_l where the specified operation will be executed. For example, in the API *tf.concat([t1, t2], 0)*, the second parameter specifies that the concatenation should be performed along the first dimension. The mutator function alters arg_i to ensure that the value of arg_i exceeds the input tensor range T_l.

Tensor List-Indices Mismatch: The mutator function associated with this rule requires two inputs: tensor T_l and a list indicating multiple dimensions of the input tensor where the operation is expected to be done. The mutator function alters the range of the list to ensure that the length of the list is not equal to the dimension of the input tensor.

List Indices Elements Mismatch: The mutator function associated with this rule requires two inputs: tensor T_l and a list indicating multiple dimensions of the input tensor where the operation is expected to be done. The mutator function alters the elements within the list to guarantee that they do not match the dimension of the input tensor.

[7]Due to space limitations, the details of the summarized root causes are available in the supplementary documentation at [1]

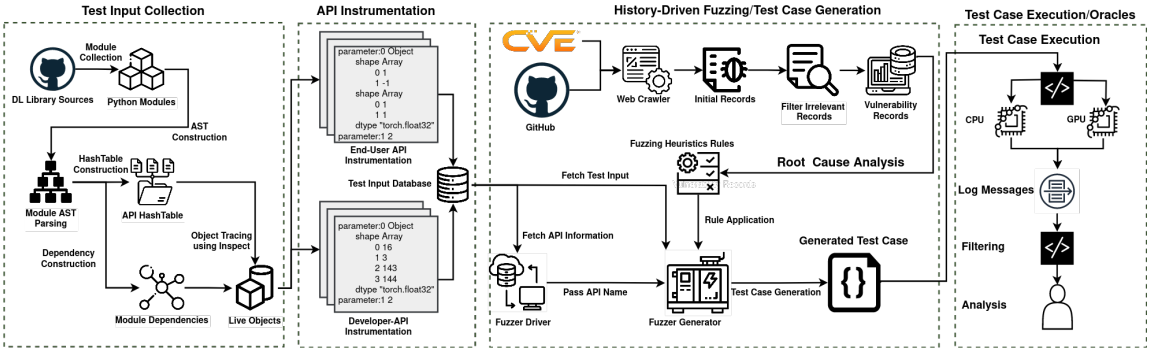


Fig. 2. Overview of our proposed Orion.

List Indices Length Mismatch: The mutator function associated with this rule requires two input lists. The mutator alters the length of these two lists to ensure that they do not have equal lengths. This results in incompatibility between the list indices and the input tensors.

In the second part, we have organized these corner case input generation for preemptive data types and tensor data types. The detailed explanation of corner case values is explained in Table 3.

Tensor Corner Case Generator Type 1: In this mutation function, the value, v in $T_j < v, s_{i\times j}^{D_{1,2,\ldots,r}}, dtype >$, of input tensors undergoes mutation based on the defined corner cases.

Tensor Corner Case Generator Type 2: In this mutation function, the first dimension, i.e., n in $T_j < v, s_{i\times j}^{D_{1,2,\ldots,r}}, dtype >$, of the input tensor undergoes mutation based on the defined corner cases.

Scalar Tensor Corner Case Generator: If the input tensor is a non-scalar tensor, this operator converts it to a scalar tensor.

Non-Scalar Tensor Corner Case Generator: If the input tensor is scalar, this function mutates it to a non-scalar tensor.

Preemptive Corner Case Generator Type 1: In this corner case generator, the parameters of type integer and real are mutated to one of the corner case values explained in Table 3.

Preemptive Corner Case Generator Type 2: In this corner case generator, the parameters of type string mutated to one of the following corner cases shown in Table 3.

List Corner Case Generator: In this corner case generator, the values of the Python list are mutated to one of the following corner cases shown in Table 3.

Mutate Tensor Data Type: In this mutator function, the type of the parameter under test is mutated to one of the allowed types within PyTorch and TensorFlow.

Orion uses the fuzzing heuristic rules summarized in Table 3 to instruct its fuzzer generator on how to perform mutations on the test inputs collected from various sources. This process is illustrated in our framework, as shown in Figure 2.

3 FRAMEWORK

Test Input Collection (Stage 1): In this initial stage, API test inputs are gathered from various sources, including library documentation, developer tests, and publicly available repositories on GitHub that utilize TensorFlow and PyTorch APIs. Further details on this data collection process can be found in subsection 3.1.

API Instrumentation (Stage 2): In the second stage, dynamic instrumentation is employed to trace execution details for each API invocation. This information encompasses parameter values and types. The collected data is then used to create a type space, API value space, and argument value space, which are crucial for the subsequent stages. More information on dynamic instrumentation can be found in subsection 3.1.

History-Driven Fuzzing/Test Case Generation (Stage 3): The third stage introduces our history-driven fuzzer, as described in subsection 3.2. Utilizing fuzzing heuristic rules, `Orion` conducts fuzzing on the test inputs retrieved from its database and generates the corresponding test cases.

Test Execution and Oracles (Stage 4): The final stage involves test execution and oracles, detailed in subsection 3.3. Test cases generated in the previous stage are executed, and their log messages are filtered based on predefined oracles. This stage includes running the test cases under two different settings: CPU and GPU. Log messages are parsed according to the specified oracles. Subsequently, manual analysis is performed to eliminate irrelevant test cases, such as those resulting from syntax errors or user errors.

3.1 Test Input Collection and API Instrumentation

Following existing work [21, 53], `Orion` performs test input collection from three sources, that is, API reference documentation, publicly available repositories, and developer tests. `Orion` then performs API instrumentation to collect various dynamic execution information from both end-user and developer APIs (that is, the type and value of each parameter). In this work, we followed FreeFuzz [53] to instrument end-user APIs for collecting their dynamic execution information. A significant challenge in automated fuzz testing of DL libraries arises from the fact that many developer APIs that are essential for library development are often overlooked. This is primarily due to their lack of comprehensive documentation and clear usage guidelines, which contrast with end-user APIs. To address the problem, we extended FreeFuzz instrumentation to make it compatible with developer APIs for the TensorFlow library. The reason we only focused on instrumenting developer APIs for TensorFlow is due to the differing API signatures between developer APIs and end-user APIs in TensorFlow. End-user APIs are accessible by importing the TensorFlow object with *import tensorflow as tf* and then invoking the desired APIs. Developer APIs, however, employ distinct signature names. Initially, developers must utilize different parent module names for the corresponding APIs at the beginning of the Python file, using the syntax *from tensorflow.python.module_name.sub_module_name.functionality import functionality*. Following this step, developers can utilize functionality anywhere in the codebase to access the desired APIs. In PyTorch, accessing developer APIs is straightforward, that is, developers can directly call these APIs by appending an underscore after . such as *Torch._*, which poses no significant challenge. Moreover, the same API instrumentation employed for end-user APIs can be utilized for developer APIs in PyTorch, i.e., there is no need for separate tooling for this. To address this challenge, we devised a solution, shown in the first box of Figure 2, as a lightweight static analyzer leveraging the Python AST module[8]. This analyzer systematically collects vital data about developer APIs within the TensorFlow library. Firstly, `Orion` clones the source code of DL libraries from GitHub and extracts all relevant Python files within the Python directory of the source codes. These modules act as intermediaries between the backend, where extensive

[8]https://docs.python.org/3/library/ast.html

computations occur, and the client code accessible to end users. Next, for each Python module file, we construct a corresponding Abstract Syntax Tree (AST). Using the AST visitor functionality provided by the Python AST module, we systematically extract the names of developer APIs present within each module. The extracted developer API names are organized into categories based on their respective parent modules. Simultaneously, we construct dependency statements for each developer's API name. To ensure the usability of the developer API names, we utilize the Python inspect module[9]. This module allows us to access live module object information associated with each developer's API name, a crucial step in facilitating test input collection. Finally, we pass the live objects for API instrumentation, covering both end-user and developer APIs.

Developer APIs are crucial for the internal testing and development of TensorFlow. Although they may not be directly exposed to end users, they play a critical role in the library's robustness and reliability. Any vulnerabilities in these APIs can indirectly affect the end-user experience. For example, bugs in developer APIs could lead to performance issues or security vulnerabilities in end-user-facing functionalities that rely on these internal components. While end users may not directly exploit these bugs, attackers can still exploit them indirectly through attacks on end-user APIs or downstream systems that rely on TensorFlow. Vulnerabilities in developer APIs could potentially be leveraged by attackers to bypass security mechanisms, compromise model integrity, or cause unintended behavior in deployed TensorFlow models.

3.2 History-Driven Test Case Generation

Before we delve into explaining the process of fuzzing, we define the following notations:

- The API name, denoted as $apiName$.
- entry, a random test input fetched from the input database associated with $apiName$.
- $numIter$, representing the total number of times the API is tested.
- A collection of fuzzing heuristic rules stored as a set of lookup tables, represented as $L_0(t), L_1(t), ..., L_{n-1}(t)$, where:
  - t corresponds to the type of current argument under test.
  - n signifies the total number of lookup tables. The lookup tables store fuzzing heuristics rules for each parameter type.

Fuzzer Driver: In the initial step, the fuzzer driver retrieves the list of all available DL APIs collected for fuzzing and iterates through this list, one API at a time. For each specific API, denoted as $apiName$, the driver passes this API name to the *Fuzzer Generator* component.

Fuzzer Generator: Algorithm 1 shows how the fuzzer generator performs fuzzing based on the constructed fuzzing heuristic rules. The fuzzer generator receives the API name from the fuzzer driver and retrieves a random test input for the API $apiName$ from the test input database. The random test input is inputted into $getValueSpace$, and the corresponding values for each parameter in the test input are received and assigned to VS. Subsequently, the generator iterates through all the parameters of VS. Please note that the fuzzer generator processes parameters one at a time, mutating the current parameter before moving on to the next. After completing the mutation for an argument, Orion maintains the state of that parameter and proceeds to the next one. It is important to note that this sequential mutation is guided by the fuzzing heuristic rules. For example, if the corner case generator is instructed to create test cases

[9]https://docs.python.org/3/library/inspect.html

Algorithm 1 History-Driven Fuzzer Generator

```
1:  procedure ORION(apiName, numIter)
2:      lookupTables = L_0(t), L_1(t), ..., L_{n-1}(t) ▷ Where n is the number of mutation operations and t is the argument type
3:      for i in |numIter| do
4:          entry = getRandomTestInput(apiName)          ▷ Get a random test input for the given API from database
5:          VS = getValueSpace(entry)                    ▷ Convert the test input to value space for fuzzing operation
6:          for arg in VS do
7:              argType = GetArgType(arg)                ▷ Get the type of the current argument
8:              heuristicRules = lookupTables[argType]   ▷ Get all applicable rules based on the type of the parameter
9:              for rule in heuristicRules do
10:                 if DoTypeMutation() then
11:                     MutateType(arg)                  ▷ Perform type mutation in a random fashion
12:                 end if
13:                 Mutate(arg, rule)                    ▷ Perform fuzzing on the current parameter according to the current rule
14:             end for
15:         end for
16:     end for
17: end procedure
```

related to *Tensors Dimension Mismatch*, the algorithm will only mutate the current tensor and its corresponding integer parameter representing the dimension of the tensor.

For the current parameter under test, denoted p_j, the generator identifies its type(t), and then the generator extracts a list of fuzzing heuristic rules available specific to that parameter type from *lookupTable*. Next, `Orion` iterates through all available fuzzing heuristic rules in the current lookup table and applies each rule, denoted r_i, to the current parameter p_j using its mutation function. To illustrate, if p_j represents a tensor, the generator systematically applies all rules specifically designed for tensors. This iterative process continues until all rules have been applied, ensuring a comprehensive exploration of the API's parameter space and corresponding input values (argument space).

Test Case Generator: During this phase of the fuzzer generator, after the test input has undergone fuzzing based on the extracted fuzzing heuristic rules, the resulting mutated test input is transformed into a Python test case. This transformation involves the inclusion of all necessary information in the test case. This information encompasses error-handling statements, statements for both CPU and GPU computations, and alignment of the test case with specific hardware configurations and computational requirements necessary for accurate execution. Once the test case is generated, the fuzzer generator submits it to execution.

3.3 Test Case Execution

In this stage, the generated test case is executed to identify potential security vulnerabilities. The log message of the generated test case is automated parsing to eliminate irrelevant execution logs, such as syntax errors and timed-out cases. To ensure the accuracy of our findings and minimize potential bias in the results, we also perform a manual examination and analysis of the output from the executed test cases. During this analysis, we consider the following oracles.

Crash Oracle: A *Crash* is defined as an event where the executed test case either halts the Python interpreter or triggers a runtime error. These issues can sometimes be induced by invalid inputs generated during the fuzzing process. We use a set of regular expression rules, made up of a set of crash-related keywords, to filter out non-crash-related

log messages generated during the test case execution phase. The keywords are as follows: *Aborted (core dumped)*, *Assertion failed*, *Floating point exception (core dumped)*, *Segmentation fault*, *Check failed*, and *Bus error (core dumped)*. The percentage of filtration is 57.8% and 10% for Pytorch and TensorFlow, respectively. In other words, 57.8% and 10% of reported security vulnerabilities are not related to crashes and are filtered out using our approach. Please note that we only use this approach to filter crash-related vulnerabilities from non-crash-related ones. This helped us a lot during vulnerability triage.

Differential testing: This oracle operates by executing the generated test case under two distinct configurations: one on CPU and the other on GPU. We performed differential testing on 135 test cases that we reported to the PyTorch and TensorFlow communities. We manually analyzed 135 log messages to find any discrepancies between generated results under CPU and GPU settings. Please note that we only did differential testing on the latest releases of PyTorch and TensorFlow including 1.13.1 and 2.13.0. The time cost of the manual analysis is approximately 24 hours for 135 test cases.

4 EXPERIMENTAL SETUP

4.1 Research Questions

To evaluate the performance of `Orion`, we design experiments to answer the following research questions:

- **RQ1**: What is the performance of Orion compared to the four baselines?
  - **RQ1.1**: What is the API coverage rate of Orion compared to the existing DL fuzzers?
  - **RQ1.2**: What is the performance of Orion compared to traditional DL fuzzers?
  - **RQ1.3** What is the performance of Orion compared to LLM-based DL fuzzers?
- **RQ2**: What is the performance of `Orion` in detecting new vulnerabilities?
- **RQ3**: What is the contribution of `Orion`'s fuzzing heuristic rules in detecting vulnerabilities?

4.2 Subject DL libraries

In this paper, we select two widely recognized and extensively used DL libraries, namely TensorFlow and PyTorch, as our primary experimental subjects. For RQ1 and to compare `Orion` versus traditional DL fuzzers, we have used earlier versions of TensorFlow (specifically, releases 2.3.0 and 2.4.0) and PyTorch (releases 1.7.0 and 1.8.0) as the subjects of our evaluation. Regarding comparison with LLM-based fuzzers, we use their reported releases for TensorFlow and PyTorch namely releases 2.10.0 and 1.12.0, respectively. For RQ2 RQ3, we have utilized the most recent releases of TensorFlow and PyTorch, namely, releases 2.13.0 and 1.13.1, respectively.

4.3 Testing Environment

Our machine is equipped with Intel(R) Core(TM) i7-10700F CPU @ 2.90GHz, NVIDIA GTX 1660 Ti GPU, 16GB of RAM, and Ubuntu 22.04. We run `Orion` and the baseline fuzzers in separate conda environments. For each release, we created an isolated conda environment and installed the all required dependencies required by each fuzzer. To manage our system resources and make the comparison fair, we run one tool at a time. We also run our proposed fuzzer `Orion` **1000** times for each API. Please note that the fuzzing budget for all tools is set to 60 seconds to make the comparison fair and unbiased.

4.4 Baseline Approaches

In this paper, we select five state-of-the-art fuzzing tools as baseline techniques, three conventional tools, and two LLM-based tools.

FreeFuzz [53]: is a DL fuzzer that performs fuzz testing on TensorFlow and PyTorch libraries for bug detection. We reuse the replication package of FreeFuzz for comparison and adopt the recommended configurations in our experiments. We configured FreeFuzz to be executed with **1000** iterations for each API. We configured it to perform value and type mutation.

DeepRel [21]: extends FreeFuzz by using test inputs from one API to test other related APIs that share similar input parameters. We carefully analyzed the DeepRel paper and found that its best configuration is 1 iteration with $top\_k = 5$. We also run DeepRel **1000** times for each API.

DocTer [55]: is a documentation-driven fuzz testing framework that generates inputs for DL APIs based on specifications mined from API reference documentation. We run the publicly available DocTer implementation shared by the authors in our comparison experiments with the suggested configurations. DocTer's execution involved **1000** iterations for each API. We carefully studied its paper and realized that conforming inputs are the best setting for bug detection. [10]

TitanFuzz [19]: is a cutting-edge DL fuzzer, that harnesses the power of Large Language Models (LLMs) for conducting API-level fuzzing on TensorFlow and PyTorch. TitanFuzz uses four different groups of mutation operators for evolutionary fuzzing including argument, suffix, prefix, and method.

AtlasFuzz [20]: is the extension of TitanFuzz where it uses historical bug data collected from the open source to guide their fuzzier generator with large language models. Note that, different from `Orion`, it uses historical bug data to generate unusual programs to expose critical bugs on TensorFlow and PyTorch releases.

In this paper, we directly use the reported vulnerabilities in the replication package of TitanFuzz and AtlasFuzz for comparison against Orion. In other words, we did not run TitanFuzz and AtlasFuzz for test case generation. The reason is the incompatibilities that arise between the specific DL releases and their corresponding CUDA configurations. To illustrate this point, consider TensorFlow 2.3.0, which requires the use of CUDA version 10.1, as specified in the official TensorFlow documentation[11]. In contrast, TitanFuzz and AtlasFuzz rely on PyTorch 1.12.1 to execute their model, a requirement documented in their replication package where Torch-1.12.1 is specified. However, PyTorch 1.12.1 requires CUDA 10.2 for GPU support, as detailed in the PyTorch release history[12]. Attempting to run TitanFuzz and AtlasFuzz on TensorFlow 2.3.0, for instance, leads to a *RuntimeError: CUDA error: no kernel image is available for execution on the device*. We observe similar incompatibility issues on PyTorch releases.

4.5 Evaluation Criteria

Following existing work [53, 55], we use the following evaluation criteria to evaluate the performance of `Orion` and the baselines:

Number of covered APIs. The number of covered APIs is a good indicator of how effective a fuzzer can be. The more APIs are covered, the more vulnerabilities will be discovered and fixed.

Number of vulnerabilities in earlier releases. We also apply `Orion` on the earlier releases of TensorFlow and PyTorch including 2.3.0, 2.4.0, 1.7.0, and 1.8.0, respectively, to compare its performance with the baseline approaches, that is,

[10] Please note that there is another setting called violating input, though, the number of detected bugs by Conforming inputs is higher than violating inputs.
[11] https://www.tensorflow.org/install/source
[12] https://pytorch.org/get-started/previous-versions/

FreeFuzz, Docter, and DeepRel. Specifically, we use detection rate and fixed rate as two metrics for the comparison. The detection rate indicates the detection performance of the fuzzer while the fixed rate indicates how many vulnerabilities are fixed in the latest releases for each library. We consider the release 2.13.0 and 1.13.1 as the latest releases for TensorFlow and PyTorch respectively.

Number of new detected vulnerabilities. The primary objective of this study is to uncover new vulnerabilities in the latest releases of TensorFlow and PyTorch, thus we also report the number of new vulnerabilities detected for the studied DL libraries. The new releases include 2.10.0, 2.11.0, and 2.13.0 for TensorFlow and 1.12.0 and 1.13.1 for PyTorch.

Contribution of Fuzzing heuristics rules. We also measure the contribution of the fuzzing heuristic rules introduced in this paper to the detection of new vulnerabilities in the latest releases of TensorFlow and PyTorch.

5 RESULT ANALYSIS

5.1 RQ1: Comparison with Baselines

To evaluate the performance of `Orion` against the five baselines, we compare the number of covered APIs by each tool across TensorFlow and PyTorch, the detection performance of `Orion` against the traditional DL fuzzers, and the detection effectiveness of `Orion` against LLM-based DL fuzzers.

5.1.1 Approach. **RQ1.1: API coverage:** Regarding TensorFlow, to be able to run the test cases effectively and perform instrumentation on the developer APIs, we build it from source by the default configuration as suggested in its reference manual[13] on the release 2.4.0. After building the library, we simply run all Python files within *tensorflow/python* directory ending with **.py*. Similarly, we build PyTorch from the source using the default configuration on 1.8.0. Since building PyTorch is computationally expensive, we perform compilation in parallel with four threads. Then we simply run test cases written in Python in this directory[14].

RQ1.2: Comparison against the traditional DL fuzzers: In comparison with the baseline fuzzers, we provide details on the number of vulnerabilities that were detected and fixed across different versions of TensorFlow and PyTorch. To perform this evaluation, we executed the tools in a range of releases, specifically versions 2.3.0 and 2.4.0 for TensorFlow and versions 1.7.0 and 1.8.0 for PyTorch. In this paper, we have considered releases 2.13.0 and 1.13.1 as the most recent versions for TensorFlow and PyTorch, respectively. It is important to note that when it comes to developer APIs, we have exclusively displayed the results for FreeFuzz since the four other fuzzers are not compatible with developer APIs. When it comes to the number of APIs tested, we maintain consistency across FreeFuzz, DeepRel, and Orion, both for TensorFlow and PyTorch, to ensure a fair comparison. This uniformity is achieved by employing the same number of APIs for all three fuzzers. This approach is adopted because all of these fuzzers rely on the MongoDB database to store their test inputs. As for DocTer, we utilize their proprietary seed database that is included in their replicated package.

RQ1.3: Comparison against the LLM-based DL fuzzers: To compare `Orion` with LLM-based DL fuzzers, we use the same releases and reported vulnerabilities mentioned in their replication package. The rationale is the compatibility issue of LLM-based fuzzers with previous releases of TensorFlow and PyTorch (see Section 4.4). Hence, we use TensorFlow 2.10.0 and PyTorch 1.12.0 for the comparison.

5.1.2 Results. **API Coverage Results**. Table 4 shows the number of APIs covered by the five fuzzers. Please note that the analysis is based on `Orion` and AtlasFuzz which is the state-of-the-art DL fuzzer. In the case of PyTorch, TitanFuzz covers 1,329 APIs while `Orion` covers 1,751 representing a significant 31% improvement over AtlasFuzz. Similarly, for

[13] https://www.tensorflow.org/install/source
[14] https://github.com/pytorch/pytorch/tree/master/test

Table 4. Statistics of covered APIs by each DL fuzzer. The percentage of improvements is computed relative to Orion.

	PyTorch		TensorFlow		
	End-User APIs	% Improvement	Developer APIs	End-User APIs	% Improvement
FreeFuzz	470	272.5%	-	688	486.8%
DocTer	498	251.6%	-	911	343.1%
DeepRel	1071	63.5%	-	1902	112.2%
TitanFuzz	1329	31.7%	-	2215	82.2%
AtlasFuzz	1377	27.2%	-	2309	74.83%
Orion	1751	-	2824	1213	-

TensorFlow, AtlasFuzz covers 2,215 APIs while Orion covers 4,037 APIs, which is an impressive 74.8% improvement over AtlasFuzz. This is because Orion covers both end-user and developer APIs. In total, Orion covers 60.5% more API compared to AtlasFuzz. The reason for the high number of covered APIs for TitanFuzz and AtlasFuzz is that given a list of API names, directly generate sample runnable code for each end-user API by using LLMs and the sample code snippets for fuzz testing and test case generation. In contrast, in our approach, we begin by collecting sample runnable code from online repositories such as GitHub repositories, which presents challenges such as encountering syntax errors, version compatibility issues, and the possibility of non-runnable code snippets. Once suitable code samples are identified, they are executed to gather dynamic execution information specific to the current end-user API, which is then stored in a database for further processing. This approach inherently faces greater difficulty in collecting test inputs for end-user APIs due to the complexities associated with sourcing and validating runnable code from online repositories. Consequently, the Orion process involves additional steps and complexities, making it more challenging to cover more APIs for the end user. Our high number of developer APIs stems from our innovative test input collection method, illustrated in Figure 2. This approach involves directly executing TensorFlow's internal test cases to gather dynamic execution information for DL APIs. Since the majority of these test cases are runnable, we can significantly expand the number of test inputs and APIs covered.

Orion vs traditional DL fuzzers. Table 5 shows the detection effectiveness of Orion compared to FreeFuzz on developer APIs. As shown, Orion outperforms FreeFuzz on both TensorFlow and PyTorch releases significantly. More specifically, Orion detects 79 and 60 more vulnerabilities versus FreeFuzz. Table 6 shows the detection effectiveness of Orion compared to the baseline fuzzers. It is observable that Orion excels in terms of vulnerability detection for both end-user and developer APIs on TensorFlow 2.3.0, Orion stands out as the most effective. Orion takes the lead, detecting 137 vulnerabilities for end-user APIs, showcasing its robustness in vulnerability detection in TensorFlow 2.4.0. In terms of PyTorch 1.7.0, Orion outperforms all three tools with 5 more vulnerabilities compared to DocTer, which is the second-best fuzzer in this release. This suggests that both Orion and DocTer are highly effective fuzzers for this release. Lastly, concerning PyTorch 1.8.0, Orion exceeds FreeFuzz and DocTer in vulnerability detection, but is surpassed by DeepRel.

Figures 3 present Venn diagrams that illustrate the number of vulnerabilities detected by both Orion and traditional baseline methods. A key observation is that Orion outperforms in discovering a greater total number of vulnerabilities across both TensorFlow and PyTorch releases. Regarding traditional DL fuzzers, Orion consistently detects the highest number of overlapping vulnerabilities when compared to FreeFuzz across all releases of TensorFlow and PyTorch. In contrast, compared to LLM-based fuzzers, Orion detects fewer overlapping vulnerabilities. This difference can be

Table 5. Comparison of Orion with FreeFuzz on developer APIs. "Det" denotes the number of detected bugs and "Fixed" is the number of fixed bugs.

Tool	TensorFlow			
	2.3.0		2.4.0	
	Det	Fixed	Det	Fixed
FreeFuzz	67	66	60	60
Orion	**146**	**142**	**120**	**113**

Table 6. Comparison of Orion with the traditional fuzzers on end-user APIs.

Tool	TensorFlow				PyTorch			
	2.3.0		2.4.0		1.7.0		1.8.0	
	Det	Fixed	Det	Fixed	Det	Fixed	Det	Fixed
FreeFuzz	31	29	90	89	15	10	11	8
DeepRel	68	8	111	23	20	17	**75**	**74**
DocTer	68	56	53	42	33	32	31	30
Orion	**99**	**91**	**137**	**124**	**38**	15	34	12

attributed to the fact that `Orion` focuses on security vulnerabilities, whereas LLM-based fuzzers primarily identify general DL bugs.

`Orion` vs LLM-based DL fuzzers. Table 8 shows a comprehensive comparison between `Orion` and LLM-based DL fuzzers, namely TitanFuzz and AtlasFuzz. The results demonstrate `Orion`'s superior performance, particularly evident in TensorFlow 2.10.0 and PyTorch 1.12.0, where it exhibits a substantial improvement of 13.63% and 18.42%, respectively. Furthermore, Figure 3 provides insights into the overlap in the number of detected vulnerabilities among `Orion`, TitanFuzz, and AtlasFuzz. It's worth noting that the total number of overlaps among the three tools is quite limited. This observation underscores the fact that each tool excels in identifying different categories of vulnerabilities. Specifically, `Orion` focuses primarily on the detection of crash vulnerabilities, while TitanFuzz and AtlasFuzz focus on logical vulnerabilities. This divergence in focus results in a higher intersection of detected vulnerabilities in the case of PyTorch, as illustrated in Figure 3f, compared to TensorFlow. In summary, these findings highlight `Orion`'s superiority over LLM-based DL fuzzers in specific versions of TensorFlow and PyTorch, emphasizing its ability to excel in the detection of crash vulnerabilities and the distinct nature of vulnerabilities detected by each tool.

AtlasFuzz and Orion both use historical data to guide their corner case generator for test case generation. Compared to AtlasFuzz, Orion takes a different approach by explicitly summarizing the fuzzing patterns found in historical bug data as heuristic rules. Instead of relying on machine learning models, Orion implements symbolic methods to generate test programs based on these summarized patterns. This means that Orion uses predefined rules and logic to create test inputs that have a high likelihood of triggering bugs, without explicitly learning from the historical data in the same way as AtlasFuzz. Although both AtlasFuzz and Orion use historical bug data, they differ in how they process and utilize this information to generate test inputs for fuzzing. AtlasFuzz relies on machine learning to implicitly learn bug-triggering patterns, while Orion uses symbolic methods to explicitly summarize these patterns as heuristic rules for test program generation.

Table 7. Detection effectiveness of Orion versus DocTer with Violating Inputs(VI).

Tools	2.3.0	2.4.0	1.7.0	1.8.0
DocTer(VI)	65	55	35	20
Orion	99	137	38	34

Table 8. Comparison of Orion with the LLM-based DL fuzzers.

Tool	TensorFlow	PyTorch
TitanFuzz	22	30
AtlasFuzz	22	38
Orion	25	45
Improvement	**13.63%**	**18.42%**

Answer to RQ1: Orion covers more APIs than the selected baselines as Orion considers both end-user and developer APIs. In terms of developer APIs, Orion can detect 117% and 100 more vulnerabilities compared to its FreeFuzz extension. In terms of end-user APIs and compared to DeepRel, Orion can detect 45.58%, 23.4%, and 90% on TensorFlow 2.3.0, 2.4.0, and PyTorch 1.7.0, respectively. Regarding LLM-based DL fuzzers, Orion can detect 13.63% and 18.42% more vulnerabilities on TensorFlow and PyTorch releases.

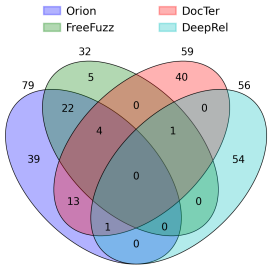
5.2 RQ2: Performance in Detecting New and Known Vulnerabilities

Table 9 presents statistics on newly detected vulnerabilities by Orion in the latest releases of TensorFlow and PyTorch. In total, Orion reports 135 vulnerabilities, 76 of which have already been confirmed by library developers, and 69 of them are new vulnerabilities. Since we report the vulnerabilities on the most recent releases for each library, at the time of writing this paper, merely seven of them have been fixed. The number of reported vulnerabilities in TensorFlow is higher than in PyTorch since the number of tested APIs in TensorFlow is higher than in PyTorch. In addition, the number of confirmed vulnerabilities within the TensorFlow library is higher since TensorFlow developers are more active in triaging the reported vulnerabilities.

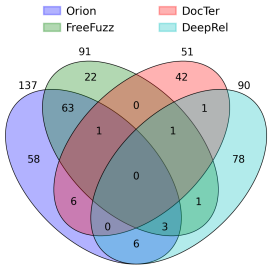
Figure 4a illustrates an example of a segmentation fault, which is one of the vulnerabilities detected by Orion in PyTorch. This vulnerability arises due to a misalignment in the input tensors for *torch.lu_unpack*. The highlighted syntax in the figure highlights this misalignment, as revealed by the rules employed in the *Guided Input Generation* process. To put it simply, the ranks of the first and second tensors should be identical. Orion derives its fuzzer generator from these mismatch rules, which are constructed based on historical vulnerabilities that share the same root cause.

Figure 4b showcases a vulnerability that Orion detected in TensorFlow. This vulnerability comes from an extremely negative value within the parameter *arg_1*, leading to a segmentation fault and subsequently crashing the Python interpreter. According to the documentation, the second parameter is expected to be a *1D* tensor with a precision of *32 bits* for integer values. However, the documentation lacks precision in specifying that large negative integer values are not allowed. Orion utilizes its extreme corner case generator to detect and expose this particular vulnerability.

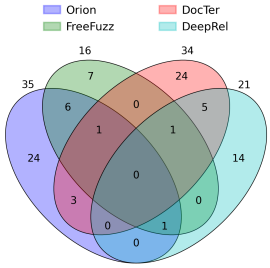
Figure 5 and Figure 6 illustrate the total number of vulnerabilities detected by Orion in the PyTorch and TensorFlow libraries, respectively. In PyTorch, *Integer Overflow* stands out as the most frequently detected bug, accounting for 14 vulnerabilities. Following closely behind is *Segfault*, which emerges as the second most detected vulnerability in the



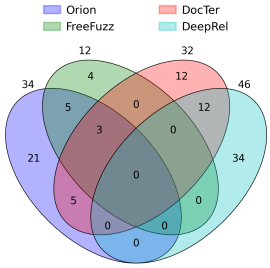
(a) TensorFlow-2.3.0



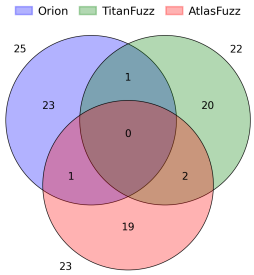
(b) TensorFlow-2.4.0



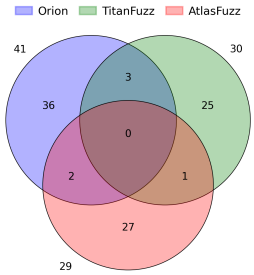
(c) PyTorch-1.7.0



(d) PyTorch-1.8.0



(e) TensorFlow-2.10.0



(f) PyTorch-1.12.0

Fig. 3. Overlap of the number of detected bugs on TensorFlow and PyTorch releases.

PyTorch library. On the other hand, in TensorFlow, *Segfault* takes the lead as the most detected vulnerability type, with 21 instances recorded. This suggests a potential susceptibility to severe memory access violations or pointer errors within the TensorFlow framework. Following *Segfault*, *Check Failed* emerges as the second most detected vulnerability, with 12 reported instances.

Table 9. The Number of new detected vulnerabilities. The number of detected unknown confirmed, and fixed bugs by Orion on TensorFlow 2.13.0 and PyTorch 1.13.1 as the latest releases.

Library	Scope	Total	Confirmed	Known	New(fixed)
TensorFlow	Developer APIs	47	28	-	28 (0)
	End-user APIs	50	21	2	19 (0)
PyTorch		38	27	5	22 (7)
Total		135	76	7	69 (7)

```
import torch                                                    Vulnerability
import numpy as np
arg_1_tensor = torch.rand([1, 0, 5, 5], dtype=torch.float64)
arg_1 = arg_1_tensor.clone()
arg_2_tensor = torch.randint(1,10,[1, 0], dtype=torch.int32)
arg_2 = arg_2_tensor.clone()
arg_3 = True
res = torch.lu_unpack(arg_1,arg_2,unpack_pivots=arg_3,)
```

(a) Segmentation fault exposed by mismatch between input tensors.

```
                                                                Vulnerability
import tensorflow as tf
from tensorflow.python.ops import gen_image_ops
arg_0 = tf.constant(-1610612736, shape=[0, 6, 6, 1], dtype=tf.bfloat16,)
arg_1 = tf.constant(-45932682421089, shape=[2], dtype=tf.int32,)
align_corners = False
out = gen_image_ops.resize_area(arg_0,arg_1,align_corners=align_corners,)
```

(b) Segmentation fault when feeding extreme corner case tensor value.

Fig. 4. Example vulnerabilities detected by Orion.

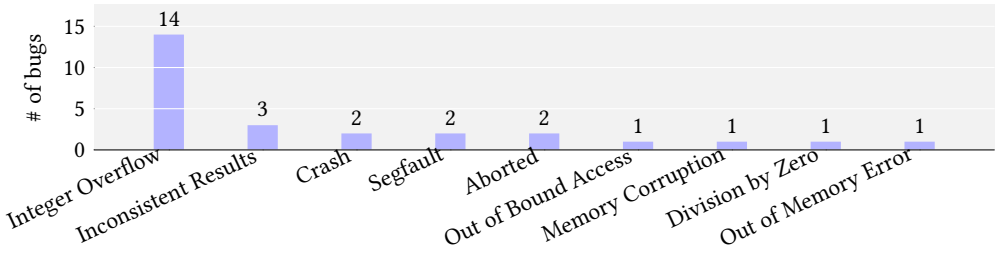


Fig. 5. Total number of confirmed bug types detected by Orion in PyTorch library.

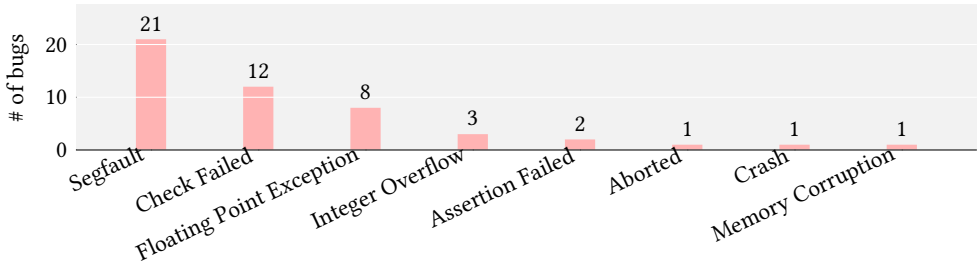


Fig. 6. Total number of confirmed bug types detected by Orion in TensorFlow library.

Figure 7 shows the mapping of root causes to the symptoms of detected vulnerabilities including new and known ones for PyTorch and TensorFlow. In the figures, the first columns show the overall fuzzing rules, while the second column specifies the specific corner cases utilized by the fuzzing rules for vulnerability detection. The last column shows the symptoms of the detected vulnerability.

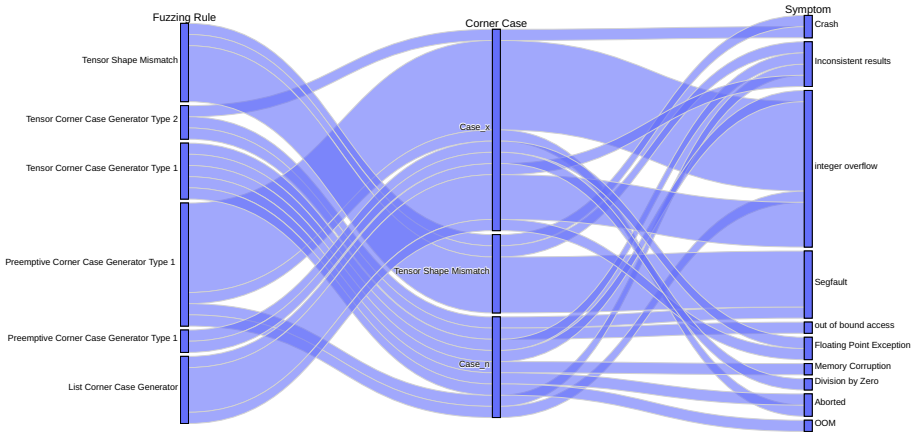
From Figure 7a which depicts the parallel plot for the PyTorch library, we can observe that there is a trace between *Tensor Shape Mismatch* and *Segfault*. A segmentation fault occurs when a program tries to access memory that it does not have permission to access, often resulting in a crash. Developers should pay close attention to ensuring that tensor shapes are consistent throughout the API execution to avoid segmentation faults. This involves a thorough shape validation to detect and handle shape mismatches appropriately. It is also observable that there is a trace between *Preemptive Corner Case Generator Type 1- List Corner Case Generator-Case_x* to *Integer Overflow*. This pattern reveals that PyTorch is vulnerable to integer overflow when parameters of type integer and list with large values are being passed to PyTorch APIs. Integer overflow occurs when a mathematical operation results in a value that exceeds the maximum representable value for the data type, leading to crashes.

The parallel plot in Figure 7b illustrates traces within the TensorFlow library. There is a significant connection between *Tensor Shape Mismatch* and *Check failed*. Unlike in the PyTorch library, in TensorFlow, vulnerabilities stemming from *Tensor Shape Mismatch* often manifest as *Check failed* issues. Additionally, there is a notable connection between *Preemptive Corner Case Generator Type 1-case_x* and both *Segfault* and *Floating Point Exception*. This implies that in TensorFlow, there vulnerabilities resulting from *case_x* tend to lead to *Segfault* and *Floating Point Exceptions*, unlike PyTorch library where *case_x* leads to *Integer Overflow*.

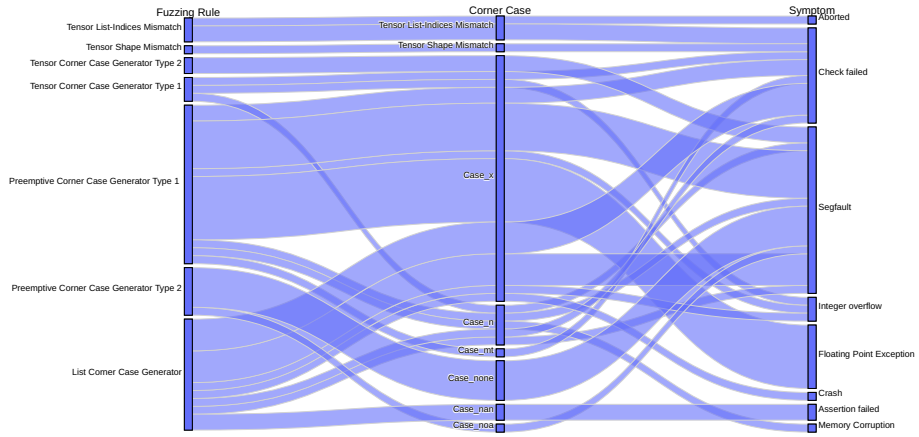
Answer to RQ2: `Orion` can detect 135 vulnerabilities, of which 76 have been confirmed by library developers. Among the confirmed vulnerabilities, 69 are unknown in the latest versions of TensorFlow and PyTorch, and 7 of them have been already fixed by library developers. The rest are awaiting further confirmation.

5.3 RQ3: Ablation Study of Fuzzing Heuristic Rules

Table 10 illustrates the contributions of fuzzing heuristic rules to the detection of vulnerabilities in TensorFlow and PyTorch. Overall, the rules categorized under *Corner Case Input Generation* demonstrate a greater impact on detecting security vulnerabilities in TensorFlow and PyTorch releases compared to those under *Guided Input Generation*. This trend can be attributed to the fact that corner case rules encompass a broader range of parameter types and vulnerable components within the API input specifications of DL APIs. Within the *Guided Input Generation* category, the *Tensor Shape Mismatch* rule exhibits the highest contribution, surpassing *Tensor List-Indices Mismatch*. This confirms that



(a) PyTorch mapping.



(b) TensorFlow mapping.

Fig. 7. Mapping of fuzzing heuristic rules to vulnerability symptoms.

the distribution of fuzzing heuristic rules contributed to the detected vulnerabilities by Orion is consistent with the real-world distribution of root causes shown in Table 3. In the category of *Corner Case Input Generation Rules*, the *case_x* and *case_n* emerge as the most effective in vulnerability detection within TensorFlow and PyTorch, confirming that the trend is consistent with the real-world distribution of root causes. This is likely because the inherent implementation of DL APIs has a weakness against zero/large/negative values. This means that when these values are provided as input to DL APIs, the back end often fails to validate these inputs properly.

Answer to RQ3: The *Corner Case Input Generation* category stands out as the most effective in terms of fuzzing heuristic rules for detecting security vulnerabilities. Among these rules, the *Large Integer Argument* rule emerges as the most effective. In comparison, the *Guided Input Generation* category is the second most effective, with the *Tensor Shape Mismatch* rule proving to be the most impactful, uncovering a total of 5 vulnerabilities.

Table 10. Contribution of Orion's fuzzing heuristics rules on detecting vulnerabilities that are confirmed by library developers including known and new instances in TensorFlow v2.13.0 and PyTorch v1.13.1.

Guided Input Generation	PyTorch	TensorFlow
Tensor Shape Mismatch	4	1
Tensor List-Indices Mismatch	-	3
Corner Cases	-	-
Case_x	16	31
Case_n	7	5
Case_none	-	5
Case_nan	-	2
Case_noa	-	1
Case_mt	-	1
Overall	27	49

6 THREATS TO VALIDITY

Internal validity. The internal validity mainly concerns whether the implementation of the tool is correct or not. To reduce such a threat, we did a code review in multiple rounds to make sure the rules were working as expected. Regarding the existing fuzzing frameworks, we did not modify their implementation and we use them directly to compare them with Orion and we use their default settings for comparison.

External validity. For this study, the external validity is on the generalizability of Orion on different DL libraries. To reduce this threat, we used two popular and widely used DL libraries including TensorFlow and PyTorch. Even though both of them use tensor-level operations and create dynamic computation graphs for DL training and testing, they have different back-end implementations. This diversity increases the validity of rules equipped with Orion. Unlike existing works which only cover end-user public APIs used only by end-users of DL libraries, we also collected dynamic execution information of developer APIs used specifically by library developers. In the future, we will extend Orion to more DL libraries. We also evaluate the effectiveness of Orion in a wide range of TensorFlow releases, including the latest and earlier releases. We also assessed the effectiveness of Orion on traditional and LLM-based DL fuzzers.

Construct validity. The main concern regarding construct validity is the evaluation criteria used for the comparison of different tools or techniques. To reduce such threats, we use four metrics for the comparison, following existing work [53, 55]. In the future, more metrics will be explored.

7 RELATED WORK

7.1 History Driven Fuzzing

History-driven fuzz testing is a testing approach [13, 40, 62, 64] that takes advantage of historical data, typically in the form of previously discovered bugs or vulnerabilities, to guide the generation of test input. Instead of randomly generating inputs, history-driven fuzz testing uses insights gained from past bugs to inform the creation of new test cases that are more likely to trigger similar issues.

Zhao et al. [64] proposed one of the latest history-driven fuzzers called JavaTailor, a novel method for generating diverse and effective test programs for Java Virtual Machine (JVM) implementations. Unlike existing techniques, which mainly focus on minor syntactic or semantic mutations, JavaTailor synthesizes test programs by integrating ingredients

extracted from historical bug-revealing test programs. Experimental results on popular JVM implementations, HotSpot and OpenJ9, demonstrate that JavaTailor outperforms existing techniques, achieving higher JVM code coverage and detecting more unique inconsistencies. Additionally, JavaTailor has identified 10 previously unknown bugs, 6 of which have been confirmed and fixed by developers. Zhang et al. [62] proposed DeltaFuzz, a fuzz testing method guided by historical version information. DeltaFuzz analyzes differences between current and previous versions to locate change points and performs change impact analysis to identify affected basic blocks. Using a genetic algorithm, it iteratively generates new test cases based on execution traces. Experiments on six open-source projects show that DeltaFuzz outperforms existing tools (AFLGo, AFLFast, and AFL), reducing the time to reach targets by 20. 59%, 30. 05% and 32. 61%, respectively. Lyu et al. [40] proposed the Probabilistic Byte Orientation Model (PBOM), which captures byte-level mutation strategies from both intra- and inter-trial history. By reusing partial-path constraint solutions from previous mutation strategies, PBOM effectively triggers unique paths and crashes. Building on this model, they introduce EMS, a history-driven mutation framework. EMS uses PBOM as a mutation operator to probabilistically determine the optimal mutation byte values based on input data. They evaluated EMS against state-of-the-art fuzzers, including AFL, QSYM, MOPT, MOPT-dict, EcoFuzz, and AFL++, in nine real-world programs. The experimental results demonstrate that EMS discovers up to 4.91 times more unique vulnerabilities than the baseline and achieves greater line coverage than other fuzzers in most programs. Chen et al. [13] proposed HiCOND, a novel approach for generating test programs in compiler testing. HiCOND leverages historical data to diversify test configurations, to produce bug-revealing and diverse test programs. By inferring ranges for test configuration options from historical data and employing particle swarm optimization, HiCOND generates effective test programs. The experimental results on GCC and LLVM show that HiCOND detects significantly more bugs compared to existing methods, with detection rates up to 145% higher. Additionally, HiCOND successfully detected 11 bugs in a practical evaluation at a global IT company.

7.2 Fuzzing DL compilers

Recently, many researchers have been attracted to the research of DL compilers [36, 37, 46]. Hence, high-performance computing is essential in DL application development. Often, DL models are compiled and optimized based on specific platforms for safety-critical application domains.

One of the first approaches to DL compiler testing was proposed by [46] who proposed TVMFuzz as a proof-of-concept application of their root cause categorization of bugs in DL compilers. TVMfuzz generates new tests based on TVM's original test suite and uncovered 8 bugs that the original suite missed.

Liu et al. [37] proposed TZer, a functional tensor compiler fuzzer that incorporates coverage guidance and utilizes a combined approach of IR (Intermediate Representation) and pass mutation. Tzer focuses on the low-level Intermediate Representation (IR) in TVM, employing both general-purpose and tensor-compiler-specific mutators driven by coverage feedback to create diverse and evolving IR mutations. Moreover, Tzer includes pass mutation along with IR mutation, utilizing the various optimization passes of tensor compilers to improve fuzzing efficiency. Experimental results show that Tzer greatly surpasses current fuzzing techniques for tensor compiler testing, achieving 75% higher coverage and producing 50% more valuable tests compared to the next best method.

NNSmith [36] is one of the most recent DL compiler fuzzers proposed in the literature. NNSmith proposed testing DL compilers (such as TVM, TensorRT, ONNXRuntime) focusing on finding bugs within the compilers that optimize DNN models. It uses a new fuzz testing approach involving generating diverse DNN models, gradient-based search to avoid floating-point exceptional values and differential testing.

Both NNSmith and Orion are fuzz-testing tools aimed at improving the reliability of DL libraries. However, they target different aspects and components within the DL ecosystem, and their methodologies, goals, and results differ significantly. In terms of goals, Orion is an API-level fuzz testing tool that primarily targets DL libraries like PyTorch and TensorFlow, focusing on both end-user and developer APIs. The primary objective of Orion is to improve fuzzing by combining guided test input generation with corner-case test inputs based on fuzzing heuristic rules derived from historical vulnerabilities. On the other hand, NNSmith focuses on testing DL compilers such as TVM, TensorRT, and ONNXRuntime. The primary objective of NNSmith is generating diverse DNN models, gradient-based search to avoid floating-point exceptional values and differential testing.

In terms of methodology, Orion proposes Guided Input Generation which Combines heuristic rules from historical vulnerability analysis to guide test generation. Another primary component of Orion is an empirical study in which it analyzes 376 vulnerabilities to construct fuzzing heuristic rules. Another important and key technology in Orion is its seed collection in which it collects test inputs for both developer APIs and end-user APIs. In contrast, NNSmith's primary component is the diverse model generation which uses lightweight operator specifications to generate diverse and valid DNN models. Additionally, NNSmith ensures that model inputs avoid floating-point exceptional values. Also, it compares outputs from different compilers to identify discrepancies and potential bugs.

In terms of impact, Orion could report 135 vulnerabilities in TensorFlow and PyTorch; 76 were confirmed by developers, with 69 previously unknown. Compared to other tools, Orion detected significantly more vulnerabilities than DeepRel and AtlasFuzz for end-user APIs and substantially outperformed FreeFuzz for developer APIs. Conversely, NNSmith Identified 72 new bugs across TVM, TensorRT, ONNXRuntime, and PyTorch. Among them, 58 bugs were confirmed and 51 were fixed by maintainers.

In conclusion, Orion offers a robust approach to fuzzing DL libraries with a focus on comprehensive API testing and leveraging historical vulnerability data. NNSmith, on the other hand, innovates in the domain of DL compilers, providing a unique method for uncovering compiler-specific bugs through model diversity and differential testing.

7.3 Testing DL libraries

Deep learning libraries have been widely used to assist users in Deep Neural Networks (DNNs) training and prediction tasks [27] such as image classification [3, 10, 12, 32, 45, 50, 58], natural language processing [17, 26, 31, 35, 38, 41, 42, 48], and software engineering [5, 14, 24, 56, 57].

Researchers have conducted numerous research studies [2, 7, 15, 18, 28, 43, 44, 52, 59, 63, 65] for testing DL libraries. One of the first steps toward testing DL libraries is the framework proposed by [44] which is a new approach that focuses on finding and localizing bugs in deep learning (DL) software libraries. It addresses the challenge of testing DL libraries by performing cross-implementation inconsistency checking to detect bugs and leveraging anomaly propagation tracking and analysis to localize the faulty functions that cause the bugs. LEMON [52] extends CRADLE by proposing a mutation-based framework where It designs a series of mutation rules for DL models to explore different invoking sequences of library code and hard-to-trigger behaviors. Another line of research is API-level testing of DL libraries [21, 34, 53, 55] where each API is considered a subject for fuzzing based on a set of predefined or random mutation rules. For example, DocTer [55] designed to analyze API documentation to extract deep learning (DL)-specific input constraints for DL API functions. FreeFuzz [53] instrumented DL API calls from different sources and performed instrumentation to trace dynamic execution information of DL APIs. Then these instruments were used for random fuzzing, where type and value mutations were applied. One limitation of FreeFuzz is that it does not consider the relational property of APIs that have similar names and parameter signatures. DeepRel [21] further extended FreeFuzz

to infer potential API relations based on API syntax and semantic information, synthesizes test programs for invoking these relational APIs, and performs fuzzing to find inconsistencies and bugs. A recently proposed fuzzer called SkipFuzz [34] uses active learning to infer the input constraints of each API function and generate valid inputs. The active learner queries a test executor for feedback, which is used to refine hypotheses about the input constraints.

Our work differs from existing DL fuzzers in several key aspects. FreeFuzz [53] and DeepRel [21] adopt approaches that involve generating random test inputs for API-level testing of TensorFlow and PyTorch. DocTer [55], on the other hand, focuses on extracting a set of input generation constraints from API reference documentation. In the context of LLM-based DL fuzzers, such as TitanFuzz [19] and AtlasFuzz [20], the emphasis is on leveraging LLMs to model API usage sequences and context information for API-level fuzzing which mostly expose general DL bugs, not security vulnerabilities. In contrast, our approach is centered on the construction of fuzzing heuristic rules based on historical security vulnerabilities. These rules are designed to mimic real-world corner cases test inputs when fuzzing DL APIs, thereby discovering critical vulnerabilities within the backend implementation of TensorFlow and PyTorch. Additionally, `Orion` employs guided input generation rules to model correlations between input arguments from DL APIs, addressing one of the main root causes of security vulnerabilities in DL libraries.

8 CONCLUSION

In this paper, we proposed `Orion`, the first step toward building a semi-automated fuzzing framework based on a set of fuzzing heuristics rules built on top of the history security vulnerabilities in TensorFlow and PyTorch. More specifically, `Orion` performs test input generation for fuzzing via mining historical data from open source, including API documentation, public repositories on GitHub, and library tests. We built the fuzzing heuristics rules based on 33 unique root causes of security vulnerabilities. We extensively evaluated `Orion` versus three traditional DL fuzzers, including FreeFuzz, DeepRel, and DocTer, as well as two state-of-the-art LLM-based fuzzers on more than 5k end-user and developer DL APIs. `Orion` reports 135 vulnerabilities, 76 of which are confirmed by the community of DL library developers. Among the 76 confirmed vulnerabilities, 69 are new vulnerabilities, 7 of them have been fixed after we reported them and the left are awaiting confirmation. In terms of end-user APIs and compared to most recent traditional DL fuzzers, i.e., DeepRel, `Orion` detects 45.58% and 90% more vulnerabilities, respectively. In comparison with the state-of-the-art LLM-based fuzzer, e.g., AtlasFuzz, `Orion` detects 13.63% and 18.42% more vulnerabilities on TensorFlow and PyTorch.

9 DATA AVAILABILITY

We share the source code of `Orion`, the bug list, and data in [1].

REFERENCES

[1] 2023. History-driven Fuzzing for Deep Learning Libraries. https://github.com/dmc1778/Orion

[2] Akshay Agarwal, Nalini Ratha, Mayank Vatsa, and Richa Singh. 2022. Exploring Robustness Connection Between Artificial and Natural Adversarial Examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 179–186.

[3] Görkem Algan and Ilkay Ulusoy. 2021. Image classification with deep learning in the presence of noisy labels: A survey. *Knowledge-Based Systems* 215 (2021), 106771.

[4] Clark Barrett, Brad Boyd, Ellie Burzstein, Nicholas Carlini, Brad Chen, Jihye Choi, Amrita Roy Chowdhury, Mihai Christodorescu, Anupam Datta, Soheil Feizi, et al. 2023. Identifying and Mitigating the Security Risks of Generative AI. *arXiv preprint arXiv:2308.14840* (2023).

[5] Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 1–6.

[6] Priscilla Benedetti, Mauro Femminella, and Gianluca Reali. 2023. Mixed-Sized Biomedical Image Segmentation Based on U-Net Architectures. *Applied Sciences* 13, 1 (2023), 329.

[7] Joey Bose, Gauthier Gidel, Hugo Berard, Andre Cianflone, Pascal Vincent, Simon Lacoste-Julien, and Will Hamilton. 2020. Adversarial example games. *Advances in neural information processing systems* 33 (2020), 8921–8934.

[8] Taejoon Byun and Sanjai Rayadurgam. 2020. Manifold for machine learning assurance. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 97–100.

[9] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. 2019. Input prioritization for testing neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 63–70.

[10] Rasim Caner Çalik and M Fatih Demirci. 2018. Cifar-10 image classification with convolutional neural networks for embedded systems. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 1–2.

[11] Nicholas Carlini. 2023. A LLM assisted exploitation of AI-Guardian. *arXiv preprint arXiv:2307.15008* (2023).

[12] Jun Chen, Han Guo, Kai Yi, Boyang Li, and Mohamed Elhoseiny. 2022. Visualgpt: Data-efficient adaptation of pretrained language models for image captioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18030–18040.

[13] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 305–316.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Seok-Hwan Choi, Jin-Myeong Shin, Peng Liu, and Yoon-Ho Choi. 2022. ARGAN: Adversarially Robust Generative Adversarial Networks for Deep Neural Networks Against Adversarial Examples. *IEEE Access* 10 (2022), 33602–33615.

[16] Darren Cofer, Isaac Amundson, Ramachandra Sattigeri, Arjun Passi, Christopher Boggs, Eric Smith, Limei Gilham, Taejoon Byun, and Sanjai Rayadurgam. 2020. Run-time assurance for learning-enabled systems. In *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings 12*. Springer, 361–368.

[17] Andrew M Dai and Quoc V Le. 2015. Semi-supervised sequence learning. *Advances in neural information processing systems* 28 (2015).

[18] Tao Dai, Yan Feng, Bin Chen, Jian Lu, and Shu-Tao Xia. 2022. Deep image prior based defense against adversarial examples. *Pattern Recognition* 122 (2022), 108249.

[19] 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.

[20] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[21] 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.

[22] Zhen Fang, Xu Ma, Huifeng Pan, Guangbing Yang, and Gonzalo R Arce. 2023. Movement forecasting of financial time series based on adaptive LSTM-BN network. *Expert Systems with Applications* 213 (2023), 119207.

[23] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: whole-proof generation and repair with large language models. *arXiv preprint arXiv:2303.04910* (2023).

[24] Park Foreman. 2019. *Vulnerability management*. CRC Press.

[25] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 385–396.

[26] Luke Gessler and Amir Zeldes. 2022. MicroBERT: Effective Training of Low-resource Monolingual BERTs through Parameter Reduction and Multitask Learning. *arXiv preprint arXiv:2212.12510* (2022).

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[28] Abigail Graese, Andras Rozsa, and Terrance E Boult. 2016. Assessing threat of adversarial examples on deep neural networks. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 69–74.

[29] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, and Song Wang. 2022. Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries. *arXiv preprint arXiv:2203.06502* (2022).

[30] Joo-Wha Hong, Yunwen Wang, and Paulina Lanz. 2020. Why is artificial intelligence blamed more? Analysis of faulting artificial intelligence for self-driving car accidents in experimental settings. *International Journal of Human–Computer Interaction* 36, 18 (2020), 1768–1774.

[31] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).

[32] Xiaowei Hu, Zhe Gan, Jianfeng Wang, Zhengyuan Yang, Zicheng Liu, Yumao Lu, and Lijuan Wang. 2022. Scaling up vision-language pre-training for image captioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 17980–17989.

[33] Yujie Ji, Xinyang Zhang, Shouling Ji, Xiapu Luo, and Ting Wang. 2018. Model-reuse attacks on deep learning systems. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 349–363.

[34] 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).

[35] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[36] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and*

Operating Systems, Volume 2. 530–543.

[37] 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.

[38] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. 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*. 101–114.

[40] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. 2022. EMS: History-Driven Mutation for Coverage-based Fuzzing.. In *NDSS*.

[41] Shervin Minaee and Zhu Liu. 2017. Automatic question-answering using a deep similarity neural network. In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 923–927.

[42] Byung-Doh Oh and William Schuler. 2022. Entropy-and Distance-Based Predictors From GPT-2 Attention Patterns Predict Reading Times Over and Above GPT-2 Surprisal. *arXiv preprint arXiv:2212.11185* (2022).

[43] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[44] 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.

[45] Mert Bulent Sariyildiz, Julien Perez, and Diane Larlus. 2020. Learning visual representations with caption annotations. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VIII 16*. Springer, 153–170.

[46] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.

[47] Zeyu Sun, Jie M Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic testing and improvement of machine translation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 974–985.

[48] Yechun Tang, Xiaoxia Cheng, and Weiming Lu. 2022. Improving Complex Knowledge Base Question Answering via Question-to-Action and Question-to-Question Alignment. *arXiv preprint arXiv:2212.13036* (2022).

[49] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 386–396.

[50] Jianfeng Wang, Zhengyuan Yang, Xiaowei Hu, Linjie Li, Kevin Lin, Zhe Gan, Zicheng Liu, Ce Liu, and Lijuan Wang. 2022. Git: A generative image-to-text transformer for vision and language. *arXiv preprint arXiv:2205.14100* (2022).

[51] Song Wang and Nachiappan Nagappan. 2021. Characterizing and Understanding Software Developer Networks in Security Development. In *The 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)*.

[52] 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. https://doi.org/10.1145/3368089.3409761

[53] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. *arXiv preprint arXiv:2201.06589* (2022).

[54] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. *arXiv preprint arXiv:2309.00608* (2023).

[55] 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.

[56] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*. PMLR, 10799–10808.

[57] Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720* (2018).

[58] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. 2022. Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917* (2022).

[59] Jiliang Zhang and Chen Li. 2019. Adversarial examples: Opportunities and challenges. *IEEE transactions on neural networks and learning systems* 31, 7 (2019), 2578–2593.

[60] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual Code Co-Evolution Using Large Language Models. *arXiv preprint arXiv:2307.14991* (2023).

[61] Jing Zhang, Qiuge Qin, Qi Ye, and Tong Ruan. 2023. ST-Unet: Swin Transformer boosted U-Net with Cross-Layer Feature Enhancement for medical image segmentation. *Computers in Biology and Medicine* (2023), 106516.

[62] Jia-Ming Zhang, Zhan-Qi Cui, Xiang Chen, Huan-Huan Wu, Li-Wei Zheng, and Jian-Bin Liu. 2022. DeltaFuzz: historical version information guided fuzz testing. *Journal of Computer Science and Technology* 37, 1 (2022), 29–49.

[63] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 132–142.

[64] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1133–1144.

[65] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 347–358.

[66] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 914–919.