

**IMPROVING THE RELIABILITY OF AI INFRASTRUCTURE
SOFTWARE WITH DATA-DRIVEN SOFTWARE ANALYTICS**

NIMA SHIRI HARZEVILI

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY OF ELECTRICAL ENGINEERING & COMPUTER
SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER
SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

FEBRUARY 2025

© Nima Shiri Harzevili, 2025

Abstract

Today, AI systems are increasingly used in safety-critical fields like transportation, finance, and robotics. While AI offers many benefits that simplify daily life, its widespread adoption has also increased threats, highlighting the urgent need for secure AI. Failing to protect AI systems against security threats could have disastrous consequences. Like traditional software, AI applications are built upon multiple layers: application and service, model, framework, library and compiler, and hardware.

In this thesis, we first conduct an empirical study to characterize and understand security weaknesses in AI frameworks. We identified Memory Leak (CWE-401) and Integer Overflow (CWE-190) as the two most prevalent bug types, with common root causes being improper validation of tensor properties and poor memory management. Next, we assess the effectiveness of five popular static analysis tools for identifying bugs in AI frameworks. Our study shows that these tools detect only a small fraction of bugs. Key limitations include lacking support for AI-specific macros/APIs, tensor data types, and computation graphs. We then evaluate dynamic analysis techniques, specifically DL fuzz testing tools, on real-world bugs in AI frameworks. Our findings show that DL fuzzers detect only 6.5% (34 out of 517) of unique bugs in our benchmark dataset. We also identify two main factors limiting the effectiveness of these tools.

Based on these findings, we developed a novel API-level DL fuzzer called Orion to address the limitations of existing fuzzers and identify new bugs in AI backend implementations. Our study confirms that most bugs stem from inadequate checks on tensor properties. In the final chapter, we characterize DL checker bugs and propose TensorGuard, an innovative tool designed to detect and repair such bugs. TensorGuard achieves an accuracy of 11.1%, surpassing the state-of-the-art bug repair baseline by 2%. We also tested TensorGuard on six months of checker-related updates (493 changes) in Google’s JAX library, successfully detecting 64 checker bugs.

Taken together, the findings from the five studies provide robust evidence that using data-driven software analytics to mine publicly available historical repositories of AI frameworks—such as code repositories and bug databases—holds immense potential for advancing the reliability of AI infrastructure software.

To my mother, my father, and my brother.

Acknowledgements

This thesis would not have been possible without the support and encouragement of many extraordinary people, to whom I am deeply grateful. I want to start by thanking my parents. Wherever I am, they always support me, encouraging me to follow my passions without worry. Thank you for being there for me at every stage of my life. My deepest gratitude goes to my supervisor, Professor Song Wang, for his constant guidance, support, and encouragement over these years. His invaluable help, both academically and personally, has been crucial on this journey. He has been more than a supervisor to me—a true friend for life. I feel incredibly fortunate to have worked with him. I am also grateful to my PhD supervisory committee: Dr. Marin Litoiu and Dr. Natalija Vlajic. Their ongoing guidance, critiques, and support have been essential to my work. Throughout my PhD research, I had the privilege of working with talented and dedicated researchers. I want to thank my colleagues and collaborators, including Professors Junjie Wang, Nachiappan Nagappan, Hung Viet Pham, Jinqiu Yang, Zhen Ming (Jack) Jiang, Alvine B. Belle, Moshi Wei, Jiho Shin, and Mohammad Mahdi Mohajer for their support and insights. Lastly, I want to thank my friends, including Jiho Shin, Moshi Wei, Mohammad Mahdi Mohajer, and Hamed Taherkhani. Your friendship and kindness made my PhD journey truly enjoyable.

Table of Contents

Abstract	ii
Dedication	iii
Acknowledgements	iv
Table of Contents	v
List of Tables	x
List of Figures	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Research Hypothesis	3
1.3 Thesis Overview	4
1.4 Thesis Contribution	7
1.5 Related Publications	8
1.6 Thesis Organization	10
2 Characterizing Bugs in AI Frameworks	11
2.1 Background and motivation	11
2.1.1 AI Ecosystem	11
2.1.2 Architecture of AI Frameworks	12
2.1.3 Significance of AI Frameworks	14
2.1.4 Bugs in AI Frameworks	16
2.2 Methodology	17
2.2.1 Data Collection	17
2.2.2 Data Labeling	19

2.3	Result Analysis	20
2.3.1	RQ1: Bug Types	20
2.3.2	RQ2: Root Causes	23
2.3.3	RQ3: Symptoms	29
2.3.4	Comparison with Traditional Software	31
2.3.5	RQ4: Fixing Patterns	32
2.3.6	Implication	33
2.4	Threats to Validity	35
2.5	Related Work	36
2.5.1	Studies on General Bugs	36
2.5.2	Studies on AI Bugs	37
2.6	Conclusion	38
3	Automatic Static Bug Detection in AI Frameworks	39
3.1	Introduction	39
3.2	Background & Motivation	41
3.3	Study Design	43
3.3.1	AI Framework Selection	43
3.3.2	Collection of Bugs from AI Frameworks	43
3.3.3	Running Static Analysis Tools	44
3.3.4	Identification of Bug Candidates	46
3.4	Experimental Results	48
3.4.1	RQ1: Detection Capability	48
3.4.2	RQ2: Detection Effectiveness	51
3.4.3	RQ3: Root Cause of Missing Real-world Bugs	53
3.5	Lessons Learned	57
3.5.1	Implication for Improving Flawfinder and RATS	57
3.5.2	Implication for Improving Cppcheck	58
3.5.3	Implication for Improving Infer	58
3.5.4	Implication for Improving Clang static analyzer	58
3.6	Threats to Validity	59
3.7	Related Work	60
3.7.1	Bug Detection	60
3.7.2	Studies on Static Detection Techniques	60
3.8	Conclusion	61

4	Benchmarking Deep Learning Fuzzers	62
4.1	Introduction	62
4.2	Approach	64
4.2.1	Selection of DL fuzzers	65
4.2.2	DL Libraries Selection	67
4.2.3	Manual Validation to Create Our Benchmark Dataset	69
4.3	Experiment Setup	71
4.3.1	Experiment Data	71
4.3.2	Configuration for Running DL Fuzzers	73
4.3.3	Configuration of DL Fuzzers	73
4.3.4	Evaluation Metrics	74
4.4	Results and Analysis	75
4.4.1	RQ1: Distribution of Bugs Types	75
4.4.2	RQ2: Effectiveness of Fuzzers	78
4.4.3	RQ3: Characteristics of Detected Bug Types	83
4.4.4	RQ4: Root Causes for Missing Bugs	84
4.5	Implications for Improving DL Fuzzers	91
4.5.1	Extend Test Input Exploration	92
4.5.2	Extend Test Case Context	92
4.6	Related Work	93
4.6.1	Fuzzing Traditional Software	94
4.6.2	Fuzzing DL Models	95
4.6.3	Fuzzing DL Libraries	96
4.6.4	Fuzzing DL Compilers	99
4.7	Threats to validity	99
4.8	Conclusion	100
5	History-Driven Fuzzing for Deep Learning Libraries	102
5.1	Introduction	102
5.2	Empirical Analysis of History Bugs	106
5.2.1	Data Collection	106
5.2.2	Construction of Fuzzing Heuristics Rules	109
5.3	Framework	113
5.3.1	Test Input Collection and API Instrumentation	114
5.3.2	History-Driven Test Case Generation	115
5.3.3	Test Case Execution	117
5.4	Experimental Setup	118
5.4.1	Research Questions	118

5.4.2	Subject DL Libraries	119
5.4.3	Testing Environment	119
5.4.4	Baseline Approaches	119
5.4.5	Evaluation Criteria	120
5.4.6	Result Analysis	121
5.4.7	RQ1: Comparison with Baselines	121
5.4.8	RQ2: Performance in Detecting New and Known Bugs	126
5.5	RQ3: Ablation Study of Fuzzing Heuristic Rules	131
5.6	Threats to validity	132
5.7	Related work	133
5.7.1	History Driven Fuzzing	133
5.7.2	Fuzzing DL Compilers	134
5.7.3	Testing DL Libraries	135
5.8	Conclusion	136
5.9	Data Availability	137
6	Taming Checker Bugs in Deep Learning Libraries	138
6.1	Introduction	138
6.2	Characterizing Checker Bugs in DL Libraries	142
6.2.1	Study Setup	142
6.2.2	DL Checker Bugs Characteristics	143
6.3	TensorGuard: A RAG-based Multi-Agent Framework to Detect and Fix DL Checker Bugs	149
6.3.1	Design of TensorGuard	150
6.3.2	Data for RAG and TensorGuard Evaluation	153
6.3.3	Experiments Setup	154
6.3.4	Experiments Results	155
6.4	Discussion	159
6.4.1	Detecting New Checker Bugs	159
6.4.2	Difference Between Checker Bugs in DL Libraries and Traditional Software	160
6.4.3	Guidelines to Avoid DL Checker Bugs	161
6.5	Threats to Validity	161
6.6	Conclusion	162
6.6.1	Data Availability	163

7	Conclusions and Future work	164
7.1	Thesis Findings and Contributions	164
7.2	Future Work	166
7.3	Closing Remarks	168
	Bibliography	170

List of Tables

2.1	Statistics of subject AI frameworks used in this chapter.	18
2.2	Subcategories of Memory. ML denotes Memory Leak, NPD is Null Pointer Dereference, IL is Infinite Loop, and UAF is Use After Free.	23
2.3	Subcategories of Numeric. IO is Integer Overflow, IP is Insufficient Precision, DZ is Division By Zero, and IU is Integer Underflow. . . .	23
2.4	Subcategories of Data Type Errors. LVTAP is Lack of Validating Tensors and Arrays Property, IVRI is Integer Variable Range Issue, UIDT is Using Improper Data Type, LOC is Lack of Overflow Checking, FVRI is Float Variable Range Issue, and ONT is Out of Bound Nanosecond for Timestamp.	24
2.5	Subcategories of Memory Errors. IMM is Improper Memory Management, IMA is Invalid Memory Access, and VDTC is Very Deep Tensors and Computation Graphs, and UR is an Uninitialized Resource. . . .	28
2.6	Subcategories of API Errors. AM is API Misuse, MP is Malicious Parameters, TPLI is Third Party Library Issue and UWA is Using the Wrong API.	30
2.7	Subcategories of Add Checkers. ACTAP is Add Checker for Tensors and Arrays Property, ACO is Add Checker for Overflow, and ACNP is Add Checker for Null Pointer Dereference.	35
3.1	Characteristics of bug fixing commits (BFCs) used in this chapter. . .	44
3.2	Characteristics of tools based on different program analysis factors. .	45
3.3	Distribution of reported warnings.	49
3.4	Distribution of bug types reported by the four static bug detectors. The warnings generated by the Clang static analyzer do not have a CWE number, so we exclude it in this table. Numbers in the brackets are false positive warnings. The numbers outside of the brackets are the total number of generated warnings.	51

3.5	Performance of the subject static analysis tools across four AI frameworks.	52
3.6	Bugs detected by Flawfinder and RATS.	52
4.1	Characteristics of DL fuzzers used in this chapter.	66
4.3	Distribution of bug categories for each DL library.	72
4.4	Statistics on the number of overlapping APIs for each DL fuzzer on PyTorch and TensorFlow.	72
4.2	Characteristics of our benchmark dataset. The numbers separated by / are shown GitHub issues/ and commits as two different sources in our dataset. The numbers in parentheses show the total number of unique records. The final real-world bug data are from PyTorch releases including 2.0.0, 2.0.1, 2.1.0, and TensorFlow releases including 2.11.0, 2.12.0, 2.13.0, 2.14.0.	72
4.5	Number of bugs in our benchmark datasets. The numbers inside the parenthesis show the percentage of bugs that relate to tensor calculation per release.	77
4.6	Time cost of running each DL fuzzer averaged over 5 iterations (in terms of CPU hours).	81
4.7	Statistics of the generated test cases averaged on 5 iterations and different releases of PyTorch (PT) and TensorFlow (TF).	82
4.8	The number of true bugs detected by DL fuzzers.	82
4.9	Performance of DL fuzzer in detecting each category of bug type.	83
4.10	Summary of root causes of missing bugs of the subject DL fuzzers. These root causes are summarised as trigger mechanisms where the fuzzers fail to detect the bugs.	86
5.1	Statistics of our curated dataset used for empirical study. The numbers in parenthesis are the total number of unique records for issues and CVE.	108
5.2	The notations we used in this chapter.	109
5.3	Summary of constructed fuzzing heuristic rules and their corresponding distribution for PyTorch and TensorFlow.	110
5.4	Statistics of covered APIs by each DL fuzzer. The percentage of improvements is computed relative to Orion.	122
5.5	Comparison of Orion with FreeFuzz on developer APIs. “Det” denotes the number of detected bugs and “Fixed” is the number of fixed bugs.	125
5.6	Comparison of Orion with the traditional fuzzers on end-user APIs.	125
5.7	Detection effectiveness of Orion versus DocTer with Violating Inputs(VI).	126
5.8	Comparison of Orion with the LLM-based DL fuzzers.	126

5.9	The Number of new detected bugs. 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.	127
5.10	Contribution of Orion’s fuzzing heuristics rules on detecting bugs that are confirmed by library developers including known and new instances in TensorFlow v2.13.0 and PyTorch v1.13.1.	131
6.1	Distribution of root causes of DL checker bugs.	144
6.2	Distribution of symptom of DL checker bugs.	147
6.3	Distribution of conditions types.	148
6.4	Distribution of fixing elements.	148
6.5	Prompt template for bug detection agent (COT).	151
6.6	Prompt template for bug detection agent (Zero Shot).	152
6.7	Prompt template for bug detection agent (Few Shot).	152
6.8	Prompt template for root cause analysis agent.	152
6.9	Prompt template for patch generation agent.	153
6.10	Statistics of data used for the experiments.	153
6.11	Performance of TensorGuard in detecting DL checker bugs using different prompting strategies.	155
6.12	Performance of TensorGuard versus <i>AutoCodeRover</i> in fixing DL checker bugs.	158
6.13	Performance of TensorGuard on detecting and fixing new checker bugs in JAX library.	159

List of Figures

1.1	Thesis roadmap.	5
2.1	AI ecosystem.	13
2.2	Architecture of AI frameworks.	14
2.3	The taxonomy of bug types studied in this work.	21
2.4	The distribution of bugs in the subject AI frameworks.	21
2.5	Distribution of root causes across AI frameworks.	26
2.6	Taxonomy of root causes in AI frameworks.	27
2.7	Distribution of symptoms across AI frameworks.	30
2.8	Mapping of root causes to symptoms.	32
2.9	Taxonomy of fixing patterns in AI frameworks.	34
2.10	Distribution of fixing patterns across different AI frameworks.	35
3.1	This figure shows two examples of buffer overflow from the TensorFlow library and Linux kernel.	42
3.2	An example of bug fixing commit in TensorFlow library in which the developer has increased the thread stack size from 2048 to 8192 to prevent buffer overflow bug. Figure 3.2b shows the generated warning by Flawfinder in which there is a possible buffer overflow at line 47.	47
3.3	The top 10 bug types in our dataset.	50
3.4	An example of heap buffer overflow in TensorFlow library.	53
3.5	An example of hard-to-detect buffer overflow bug in TensorFlow library.	56
4.1	Overview of dataset construction and evaluation of DL fuzzers.	65
4.2	Top bug types in PyTorch library.	75
4.3	Top bug types in TensorFlow library.	75
4.4	Distribution of detected bug types.	85
4.5	An example of Runtime Error in PyTorch library missed by the subject DL fuzzers. This bug should be triggered by <i>Modeling Parameter Space</i>	88

4.6	An example of <code>NotFoundError</code> in TensorFlow library missed by the subject DL fuzzers. This bug should be triggered by <i>Modeling Specific Data Types</i> in tensor creation.	88
4.7	An example of <code>Check Failed</code> in TensorFlow library detected by ACETest and missed by TitanFuzz and FuzzGPT. This bug should be triggered by generating <i>Inputs with Incompatible Shapes</i>	89
4.8	An example of <code>Runtime Error</code> in PyTorch library missed by the subject DL fuzzers. This bug should be triggered by calling <i>torch.nn.LSTM</i> and <i>torch.nn.jacrev</i>	89
4.9	An example of <code>Assertion Error</code> in PyTorch library missed by the subject DL fuzzers. This bug should be triggered by calling <i>torch.compile</i> and <i>torch.nn.functional.interpolate</i> together with the optional parameter of the mode set to "bilinear".	89
5.1	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 <i>gen_array_ops.lower_bound</i> which eventually resulted in the detection of a segmentation fault.	104
5.2	Overview of our proposed Orion	113
5.3	Overlap of the number of detected bugs on TensorFlow and PyTorch releases.	124
5.4	Example bugs detected by Orion	128
5.5	Total number of confirmed bug types detected by Orion in PyTorch library.	128
5.6	Total number of confirmed bug types detected by Orion in TensorFlow library.	129
5.7	Mapping of fuzzing heuristic rules to bug symptoms.	130
6.1	An example of a checker bug (#76715) in PyTorch library. The original checker failed to validate if the index <i>k</i> is less than the size of the dimension of the input tensor.	139
6.2	Taxonomy of DL checker bugs.	142
6.3	Architecture of TensorGuard	150

Chapter 1

Introduction

1.1 Motivation

Improving software reliability is now essential for guaranteeing the security and integrity of software systems [1, 2, 3, 4]. Given the increasing reliance on software for many critical processes such as financial transactions, autonomous driving [5, 6], and mission-critical systems [7, 8], the frequency of bugs pose serious risks [9, 10]. Malicious entities can exploit software bugs to gain unauthorized access, compromise sensitive information, or disrupt services if they go undetected or ignored [8]. There are multiple steps in enhancing software reliability including understanding and characterizing bugs [11, 12, 13, 14, 15, 16, 17], detecting bugs [18, 19, 20, 21, 22, 23, 24, 25, 26], and eventually fixing them [27, 28, 29]. *Although significant progress has been made in improving reliability in traditional software over the past two decades, there remains a notable gap in AI infrastructure software since it is an evolving and complex area that presents unique challenges compared to traditional software.* In particular, there are a couple of challenges when it comes to improving the reliability of AI infrastructure software:

- **C_1 : Gap Between Bugs in Traditional Software and AI Infrastructure Software:** Enhancing software reliability involves several key steps, including understanding and characterizing bugs and weaknesses [11, 12, 13, 14, 15, 16, 17], detecting bugs [18, 19, 20, 21, 22, 23, 24, 25, 26], and ultimately resolving them [27, 28, 29]. *While significant advancements have been achieved in enhancing the reliability of traditional software over the past two decades, a considerable gap persists in addressing the reliability of AI infrastructure software. This is due to its evolving nature and the unique challenges it poses compared to conventional software systems.*

- **C_2 : Limitations in Static Analysis of Bugs in AI Infrastructure Software:** Previous research has examined static analysis tools on traditional projects from different aspects [30, 31, 32, 33, 34, 35]. However, it has not been investigated if the findings of these studies on conventional software apply to AI infrastructure software. Understanding the benefits and drawbacks of static analysis tools currently in use will direct future work toward designing new techniques for detecting AI-specific bugs.
- **C_3 : Limitations of API-level Fuzz Testing of AI APIs:** In recent years, fuzz testing [36, 37, 38, 39, 40, 41, 42, 43], which detects bugs via generating malformed inputs, has been adopted to test AI frameworks [44, 45, 46]. Although these tools have been demonstrated to be effective in detecting bugs in AI frameworks and outperforming their prior work in finding more or different bugs, there remains a gap between the evaluated results and the tools’ practical effectiveness. Specifically, previous lines of research failed to comprehensively evaluate the effectiveness of tools, the scope of bug detection, and their limitations against bugs in the ground truth AI framework [44, 45, 46, 47, 48]. This leaves an unanswered question: *How effectively and thoroughly can these DL fuzzers find bugs in practice?* To our knowledge, there is no study in the literature to evaluate DL fuzzers for detecting real-world bugs based on an extensive ground-truth bug dataset.
- **C_4 : Limitation of API Level DL Fuzzing Tools in Corner Case Value Generation:** Based on insights gained from benchmarking existing API-level DL fuzzers, we identified three major limitations. First, existing DL fuzzers [44, 45, 46, 47, 48] primarily treat the parameters of DL APIs independently during API-level fuzz testing. However, our empirical study reveals that some bugs arise from specific combinations of input arguments. For instance, a mismatch between input arguments is one of the major root causes of reported bugs. Second, conventional API-level DL fuzzers [44, 46, 45, 47, 48] face two key challenges in generating corner case test inputs. Their heuristic rules for generating such inputs are often *random* and *unguided*. As a result, the generated values may fail to accurately replicate real-world bugs or the tainted inputs provided by external attackers. Since API calls often depend on specific data formats, constraints, or patterns, random values may not adhere to these requirements, potentially leading to the omission of critical bugs. Third, both conventional and LLM-based DL fuzzers [44, 46, 45, 47, 48] predominantly focus on testing end-user DL APIs. This emphasis stems from the fact that compared

to end-user APIs, developer APIs in DL libraries often lack comprehensive documentation and usage guidelines.

- **C_5 : Challenges in Repairing Bugs within AI Infrastructure Software:** Research on automatically detecting and fixing checker bugs in software systems has shown promising results in improving software reliability [49, 50, 51, 52, 53, 54]. However, while significant progress has been made in this area for conventional software, similar issues in AI frameworks have received little attention. This lack of focus stems primarily from the fundamental differences in how AI frameworks are developed compared to traditional software [55, 56, 57]. Consequently, most existing research remains centered on checker bugs in conventional software, leaving a significant knowledge gap regarding bugs specific to AI frameworks.

1.2 Research Hypothesis

Research Hypothesis: Publicly available historical repositories of AI frameworks, such as code repositories and bug databases, provide valuable insights into the software development process of AI systems. Leveraging these repositories can enhance the reliability of AI infrastructure software by deepening the understanding of weaknesses, facilitating the detection and repair of bugs, and guiding the adoption of best practices in software engineering for AI systems.

In this thesis, we primarily collect data from two types of repositories to address the aforementioned limitations. *Code repositories* refer to version control systems such as GitHub. *Bug repositories* are databases where bugs are reported, including the Common Vulnerabilities and Exposures (CVE) database and the National Vulnerability Database (NVD). To tackle the challenges and limitations discussed in the previous section (C_1 – C_5), we propose corresponding research outcomes (O_1 – O_5) that directly address these issues.

- O_1 : We take the first step in characterizing and understanding the bugs of seven well-known AI frameworks: TensorFlow, PyTorch, Scikit-learn, Mlpack, Pandas, NumPy, and SciPy.
- O_2 : We aim to evaluate the effectiveness of five popular static analysis tools on AI frameworks. Specifically, we conduct an empirical analysis using 410 real-world bugs collected from four widely used AI frameworks to determine

how many of these bugs can be detected by static analysis tools and why they fail to identify certain real-world bugs in AI frameworks.

- O_3 : We set out to conduct the first evaluation of state-of-the-art DL fuzzers against real-world bugs. Specifically, we first manually created an extensive DL bug benchmark dataset, which includes 517 real-world DL bugs gathered from PyTorch and TensorFlow libraries that are detectable by fuzzing, i.e., can be triggered by malformed inputs. Then we apply seven state-of-the-art DL fuzzers, i.e., FreeFuzz, DeepRel, NablaFuzz, DocTer, ACETest, TitanFuzz, and FuzzGPT, to the curated benchmark dataset by following their instructions.
- O_4 : 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.
- O_5 : We present the first comprehensive study of DL checker bugs in two widely-used DL libraries, i.e., TensorFlow and PyTorch. Using the knowledge gained via root cause analysis of checker bugs, we further propose TensorGuard, a proof-of-concept Retrieval Augmented Generation (RAG)-based LLM-based tool to detect and fix checker bugs in DL libraries via prompt engineering a series of ChatGPT prompts.

1.3 Thesis Overview

This section offers an overview of the works presented in this thesis, (the overall road map is depicted in Figure 1.1), which is organized into six main chapters. Each chapter addresses a distinct problem and is designed to be self-contained. Consequently, some overlap may occur across chapters. Additionally, each chapter includes a summary of related works relevant to its specific objective.

- **Chapter 2: Characterizing and Understanding Bugs in Traditional Software and AI Frameworks:** The application of AI frameworks has increased tremendously across many domains, including autonomous driving systems, healthcare, and critical industries. The bugs in such libraries could have irreparable consequences. However, the characteristics of software bugs in AI frameworks have not been well studied. We take the first step toward characterizing and understanding the bugs of seven well-known AI frameworks: TensorFlow, PyTorch, Scikit-learn, Mlpack, Pandas, NumPy, and SciPy. To

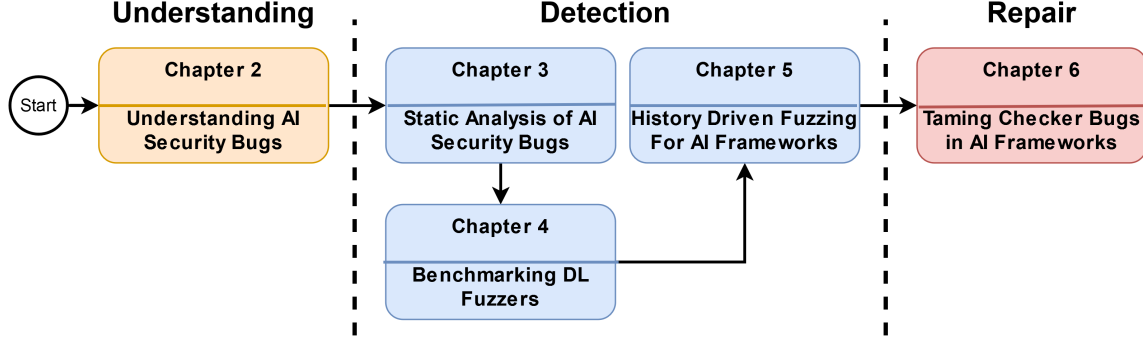


Figure 1.1: Thesis roadmap.

do so, we collected 683 bugs to explore four major factors: 1) type of bug, 2) root causes, 3) symptoms, and 4) patterns for fixing bugs in the AI frameworks studied. This study will help characterize both the commonalities and differences between bugs in AI frameworks and traditional software.

- **Chapter 3: Automatic Static Bug Detection in AI Frameworks:** Automatic detection of bugs is crucial for software quality assurance. Many static analysis tools have been proposed to help detect these bugs. However, while these tools are mainly evaluated on general software projects, their practical effectiveness and usefulness for AI frameworks remain questionable. In this chapter, we address this issue by analyzing five popular and widely used static analysis tools: Flawfinder, RATS, Cppcheck, Facebook Infer, and Clang Static Analyzer. We apply these tools to a curated dataset of 410 known bugs gathered from four popular AI frameworks: Mlpack, MXNet, PyTorch, and TensorFlow. This chapter categorizes the capabilities of these tools to better understand their strengths and weaknesses in detecting bugs in AI frameworks. Overall, the results show that static analysis tools detect a negligible percentage of the total bugs, identifying only 5 of the 410 unique bugs (0.01%). Among the tools, Flawfinder and RATS proved to be the most effective in detecting bugs in AI frameworks. We also identify opportunities for improving the effectiveness and practical application of these tools based on our findings. This chapter provides valuable insights into the scope of static analysis tools' applicability to AI frameworks and highlights their limitations.
- **Chapter 4: Benchmarking Deep Learning Fuzzers:** In recent years, the practice of fuzzing Deep Learning (DL) libraries has garnered significant attention in the software engineering community. Numerous DL fuzzers have been

proposed to test DL APIs by generating malformed inputs. Although many of these fuzzers have proven effective in detecting bugs and outperforming previous techniques, there remains a gap in benchmarking these DL fuzzers concerning their effectiveness against real-world bugs in DL libraries. Existing comparisons among these DL fuzzers primarily focus on the number of bugs detected, but they fail to provide a direct, in-depth evaluation of different DL fuzzers. In this chapter, we conduct the first evaluation of state-of-the-art DL fuzzers against real-world bugs. Specifically, we manually create an extensive DL bug benchmark dataset, which includes 517 real-world DL bugs collected from the PyTorch and TensorFlow libraries that can be detected by fuzzing—that is, triggered by malformed inputs. We then apply seven state-of-the-art DL fuzzers—FreeFuzz, DeepRel, NablaFuzz, DocTer, ACETest, TitanFuzz, and FuzzGPT—to the curated benchmark dataset, following their respective instructions. We find that these fuzzers detect only 6.5% (34 out of 517) of the unique real-world bugs in our benchmark dataset. Our analysis further identifies three prevailing factors that influence the effectiveness of these fuzzers in detecting real-world bugs. These findings highlight opportunities to enhance the performance of fuzzers in future work. Overall, this chapter complements prior studies on DL fuzzers with a comprehensive evaluation and provides a benchmark for future DL library fuzzing research.

- **Chapter 5: History-Driven Fuzzing in AI Frameworks:** Recently, DL fuzzers have been proposed for API-level testing of DL libraries. However, these fuzzers either perform unguided input generation (e.g., not considering the relationships between API arguments when generating inputs) or only support a limited set of corner-case test inputs. Furthermore, many developer APIs, which are crucial for library development, remain untested due to a lack of comprehensive documentation and clear usage guidelines—unlike end-user APIs. This makes them a more challenging target for automated testing. To address this gap, in this chapter, we propose a novel fuzzer named Orion, which combines guided test input generation with corner-case test input generation. Orion leverages a set of fuzzing heuristic rules constructed from historical data known to trigger critical issues in the underlying implementation of DL APIs. To extract these fuzzing heuristic rules, we first conduct an empirical study on the root cause analysis of 376 bugs in two of the most popular DL libraries, PyTorch and TensorFlow. We then construct the fuzzing heuristic rules based on the root causes of these historical bugs. Using these rules, Orion generates corner-case test inputs for API-level fuzzing. Additionally, we extend the seed

collection from existing studies to include test inputs for developer APIs. Our evaluation shows that Orion reports 135 bugs in the latest releases of TensorFlow and PyTorch, 76 of which were confirmed by the library developers. Among the 76 confirmed bugs, 69 were previously unknown, and 7 have already been fixed. The remaining bugs are awaiting further confirmation.

- **Chapter 6: Taming Checker Bugs in AI Frameworks:** Checker bugs in AI frameworks are critical yet not well-explored. These bugs are often concealed in the input validation and error-checking code of AI frameworks and can lead to silent failures, incorrect results, or unexpected program behavior in AI applications. Despite their potential to significantly impact the reliability and performance of AI-enabled systems built with these frameworks, checker bugs have received limited attention. The complex nature of AI frameworks, combined with the rapid pace of their development, has left a gap in understanding how these bugs manifest, propagate, and affect the overall robustness of AI systems. To fill this gap, we present the first comprehensive study of DL checker bugs in two widely-used AI frameworks, i.e., TensorFlow and PyTorch. Initially, we automatically collected a dataset of 2,418 commits from TensorFlow and PyTorch repositories on GitHub from Sept. 2016 to Dec. 2023 using specific keywords related to checker bugs. Through manual inspection, we identified 527 DL checker bugs. Subsequently, we analyzed these bugs from three perspectives, i.e., root causes, symptoms, and fixing patterns. Using the knowledge gained via root cause analysis of checker bugs, we further propose TensorGuard, a proof-of-concept Retrieval Augmented Generation (RAG)-based LLM-based tool to detect and fix checker bugs in AI frameworks via prompt engineering a series of ChatGPT prompts. We evaluated TensorGuard’s performance on a test dataset that includes 92 buggy and 135 clean checker-related changes in TensorFlow and PyTorch from January 2024 to July 2024. Our results demonstrate that TensorGuard has a high average recall (94.51%) using Chain of Thought prompting, a balanced performance between precision and recall using Zero-Shot and Few-Shot prompting strategies. Regarding patch generation, TensorGuard achieves an accuracy of 11.1%, which outperforms the state-of-the-art bug repair baseline by 2%.

1.4 Thesis Contribution

This thesis makes the following contributions:

- We conduct the first empirical study to characterize and understand bugs in AI frameworks. We show detailed taxonomies regarding the types, root causes, symptoms, and fixing patterns of bugs in AI frameworks.
- We present the first empirical study on the investigation of current static analysis tools on four widely used AI frameworks. We also identify gaps between existing static analysis tools techniques and practices on AI frameworks.
- We conduct the first empirical study to evaluate state-of-the-art DL fuzzers against real-world DL bugs, providing a complementary perspective on prior studies regarding comparing DL fuzzers.
- We present an effective approach to API-level testing of deep learning libraries. To the best of our knowledge, this is the first study to leverage historical bug data to create fuzzing heuristic rules.
- We present the first large-scale analysis to demystify checker bugs within PyTorch and TensorFlow libraries. We then introduce a novel LLM-based DL checker bug detection and repair tool, TensorGuard. Our evaluation shows that our tool outperforms the state-of-the-art.

1.5 Related Publications

Earlier versions of the work presented in this thesis have been published in the following papers, listed in chronological order. My primary contributions to these studies include collecting and processing raw data, conducting experiments, analyzing results, and writing. These efforts account for more than 80% of the work involved in these studies.

- **Nima Shiri Harzevili**, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming (Jack) Jiang, Nachiappan Nagappan. A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning. ACM Computing Surveys (**CSUR 2024**). 34 pages.
- **Nima Shiri Harzevili**, Jiho Shin, Junjie Wang, Song Wang, and Nachiappan Nagappan. Characterizing and Understanding Software Security bugs in Machine Learning Libraries. 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (**MSR 2023**). 12 pages. Acceptance Rate=37% (43/115).

- **Nima Shiri Harzevili**, Mohammad Mahdi Mohajer Moshi Wei, Hung Viet Pham, Song Wang. History-Driven Fuzzing for Deep Learning Libraries. ACM Transaction on Software Engineering and Methodology (**TOSEM 2024**). 28 pages.
- **Nima Shiri Harzevili**, Jiho Shin, Junjie Wang, Song Wang, and Nachiappan Nagappan. Automatic Static Vulnerability Detection for Machine Learning Libraries: Are We There Yet? 2023 IEEE 34th International Symposium on Software Reliability Engineering (**ISSRE 2023**). 12 pages. Acceptance Rate=25% 62/247.
- **Nima Shiri Harzevili**, Moshi Wei, Mohammad Mahdi Mohajer, Hung Viet Pham, Song Wang. Evaluating Deep Learning Fuzzers: A Comprehensive Benchmarking Study. ACM Transaction on Software Engineering and Methodology (**TOSEM 2025**).
- [Under Review](#) **Nima Shiri Harzevili**, Mohammad Mahdi Mohajer, Jiho Shin, Moshi Wei, Gias Uddin, Jinqiu Yang, Junjie Wang, Song Wang, Zhen Ming (Jack) Jiang, Nachiappan Nagappan. Checker Bug Detection and Repair in Deep Learning Libraries. (**TSE 2025**).

The following papers were published in parallel to the aforementioned publications. Although these studies are not directly related to AI infrastructure software, they explored ways to improve the reliability of deep learning APIs.

- Moshi Wei, Yuchao Huang, Junjie Wang, Jiho Shin, **Nima Shiri Harzevili**, and Song Wang. API Recommendation for Machine Learning Libraries: How Far Are We? The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (**ESEC/FSE 2022**). 12 pages. Acceptance Rate=22% 99/449.
- Moshi Wei, **Nima Shiri Harzevili**, YueKai Huang, Jinqiu Yang, Junjie Wang, Song Wang. Demystifying and Detecting Misuses of Deep Learning APIs. Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (**ICSE 2024**). 12 pages. Acceptance Rate=22% 234/1053.

1.6 Thesis Organization

We organized this thesis as follows. Chapter 2 elaborates on characterizing and understanding bugs in AI frameworks. Chapter 3 describes our study on automatic static bug detection in AI frameworks. Chapter 4 presents our study on benchmarking deep-learning fuzzers. Chapter 5 discusses our history-driven fuzzing for AI frameworks. Chapter 6 presents our work on taming checker bugs in AI frameworks. Lastly, chapter 7 concludes this thesis and discusses future research directions.

Chapter 2

Characterizing Bugs in AI Frameworks

2.1 Background and motivation

This chapter begins by exploring the AI ecosystem and its unique characteristics. Next, we delve into the AI framework layer, examining its internal architecture and operational mechanisms. We then provide background information to highlight its importance. Finally, we discuss bugs in AI frameworks, emphasizing how they differ from those in traditional software.

2.1.1 AI Ecosystem

The AI ecosystem [58], shown in Figure 2.1, is a vibrant and interconnected environment in which artificial intelligence technologies grow, innovate, and intersect in multiple areas. The AI ecosystem consists of multiple layers as follows:

Application/Service: The application layer in the AI ecosystem represents the interface between AI models and end-users or other application domains. At this level, the focus is on leveraging the trained models to solve specific problems or tasks in real-world scenarios such as transportation [59, 60], finance [61], market forecasting [62, 63], Robotics [64, 65]. The application layer encapsulates the deployment of AI models, seamlessly integrating them into software applications, websites, or other systems to deliver intelligent functionalities. The application layer serves as the bridge between the underlying AI infrastructure software and the practical user-facing aspects, embodying the tangible impact of AI solutions across various domains.

Model: The model layer in the AI stack is the core component where AI models [66, 67, 68, 69, 70] are trained and deployed to process data and generate predictions or insights. It includes algorithms and architectures like transformers, CNNs, or RNNs, which enable AI systems to understand and solve complex tasks such as image recognition or natural language processing. This layer is crucial to users because it determines the intelligence, accuracy, and performance of the AI system, enabling personalized, reliable, and scalable solutions tailored to user-specific needs. Without a robust model layer, AI applications cannot deliver meaningful or high-quality outputs.

Framework: The framework layer [71, 72, 73, 74, 75] in the AI stack is a fundamental component that provides the necessary tools and frameworks for developing, training, and deploying machine learning models. This layer encompasses the various libraries and frameworks, such as TensorFlow, PyTorch, Scikit-learn, and others, that offer pre-built functions and modules for tasks like data pre-processing, model building, and evaluation. The library layer acts as an abstraction, simplifying complex processes and allowing developers and data scientists to focus on model architecture and experimentation.

Library/Compiler: AI Library/compiler play a critical role in the efficient execution of AI models across diverse hardware platforms [76, 77, 78]. They optimize performance models, translating high-level representations into low-level code that can run on CPUs, GPUs [79], or specialized hardware like TPUs [76]. This is crucial because well-optimized AI compilers can dramatically enhance the speed and efficiency of AI applications, particularly in environments where performance is key, such as autonomous vehicles [80].

Hardware: The hardware layer in the AI stack provides the computational infrastructure required to run AI models efficiently. This layer includes processors like CPUs, GPUs, TPUs, and specialized AI chips, as well as memory, storage, and networking components. It is critical because AI workloads, particularly in deep learning, demand significant computational power for tasks like training large models and processing vast datasets.

2.1.2 Architecture of AI Frameworks

AI frameworks have been widely used to assist users in tasks such as image classification [81, 82, 83, 84, 85, 86, 87], natural language processing [88, 89, 90, 91, 92, 93, 94, 95], and software engineering [2, 96, 97, 98, 99].

The diagram in Figure 2.2 illustrates the comprehensive architecture of an AI framework, the TensorFlow library, which consists of two main components. The first

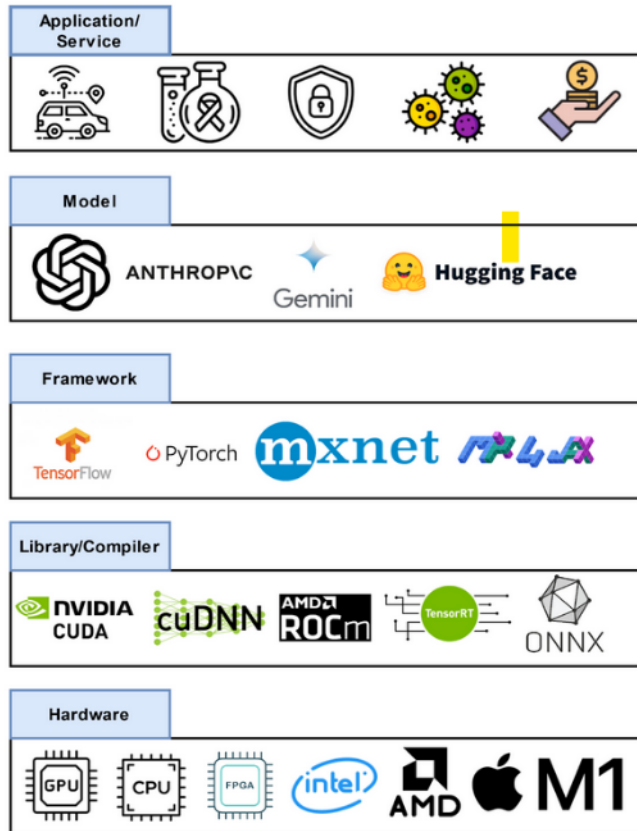


Figure 2.1: AI ecosystem.

component is the front end, where users engage high-level APIs to leverage computational capabilities in the backend. The right side of Figure 2.2 shows the layered architecture of the TensorFlow library, which consists of three main components: front-end, back-end, and hardware abstraction. The front-end handles communication with users using different programming languages such as Python, JavaScript, and C++, while the back-end implements the functionalities of TensorFlow, such as kernel implementation, where main computational operations are executed, like the *sparse* module for sparse tensor processing, or the *linealg* module responsible for linear algebra operations. The lowest level is the hardware abstraction, which refers to the library's ability to work with different types of hardware such as Central Processing Unit(CPU), Graphical Processing Unit(GPU), and Tensor Processing Unit(TPU) without the user needing to understand the specific details of how the hardware works. This allows developers to focus on creating and training models without

worrying about low-level details like memory management or device synchronization. Many popular AI frameworks like TensorFlow and PyTorch have built-in support for hardware abstraction, making it easy for developers to switch between different types of hardware or even run models on multiple types of hardware simultaneously, allowing for better resource utilization and faster model training.

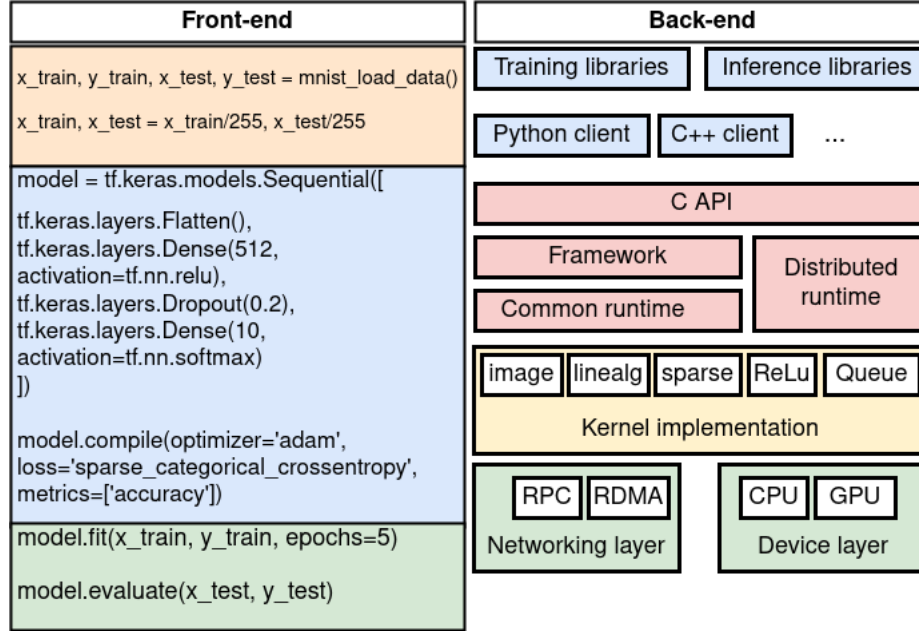


Figure 2.2: Architecture of AI frameworks.

2.1.3 Significance of AI Frameworks

Nowadays, we can witness the widespread usage of intelligent systems [81, 100, 101, 102, 103, 104, 105, 88, 106, 107] in safety-critical systems by governments, organizations, businesses. These safety critical domains are including but not limited to transportation [59, 60], finance [61], market forecasting [62, 63], Robotics [64, 65], and etc. As intelligent systems are built upon AI frameworks, these libraries are susceptible to various attacks, as highlighted in [108]. Failing to identify bugs in AI frameworks could result in catastrophic consequences, as emphasized in [109]. In the following paragraphs, we elaborate on some recent and well-known AI incidents, as reported in [109].

Medical and Health: One of the most recent incidents in this domain is found by research conducted jointly by the University of Toronto, the Vector Institute, and

MIT revealed that the input databases employed in training AI systems for chest X-ray classification introduced biases related to gender, socioeconomic status, and race. Startups affiliated with Google, such as Qure.ai, Aidoc, and DarwinAI, utilize AI to analyze chest X-rays for conditions like fractures and collapsed lungs. The study found that the training databases predominantly featured examples of white patients (67.64%), resulting in the diagnostic system exhibiting higher accuracy in diagnosing white patients compared to other demographics. Notably, black patients were only half as likely to receive recommendations for further care when necessary¹. Another notable incident is a series of malfunctions in Robotic Surgeries². Between 2000 and 2013, incidents of robotic surgeries leading to injuries and fatalities were documented in the Manufacturer and User Facility Device Experience (MAUDE) database, a repository containing both voluntary and mandatory reports of mishaps. Over these 14 years, there were 8,091 recorded malfunctions, resulting in 1,391 injuries and 144 deaths. The injuries encompassed a range of scenarios, including burns caused by sparks emitted by the machines (n=193), dislodgement of robotic arms within the patient (n=100), and situations where surgeons lost control of the machine or experienced unexpected machine shutdowns (n=52). Notably, approximately 62% of the reported injuries and deaths were attributed to system or hardware errors, while the remaining cases were ascribed to the inherent risks associated with surgery or human error.

Transportation: One remarkable incident in sector happened in India where an autonomous train crashed into a crowd in Delhi due to malfunctions in the break system³. The train, created by Huandai Rotem and overseen by the Delhi Metro Rail Corporation (DMRC), underwent testing six days before its scheduled public inauguration. During the test, it failed to halt at the designated endpoint, proceeding to pass through a wall. Fortunately, no injuries were reported. In response to the incident, the DMRC took action by terminating the employment of four individuals who had neglected to inspect the brake system before conducting the test. The next incident pertains to the Tesla Autopilot system, which encountered challenges in accurately identifying red traffic lights. Specifically, the system mistakenly recognized red letters on a flag as red traffic lights⁴.

These incidents underscore the critical need for caution in the development and deployment of intelligent systems, particularly in sensitive domains like medicine and health. As the deployment of intelligent systems heavily relies on AI frameworks,

¹<https://incidentdatabase.ai/cite/81/>

²<https://incidentdatabase.ai/cite/5/>

³<https://incidentdatabase.ai/cite/31/>

⁴<https://incidentdatabase.ai/cite/97/>

developers must prioritize robust testing, continuous monitoring, and proactive measures to identify and mitigate potential risks within AI frameworks.

2.1.4 Bugs in AI Frameworks

Much like traditional software, AI frameworks are prone to bugs because AI frameworks share similar bug patterns and deployment patterns at higher levels of granularity. However, when we finer the level of granularity, AI frameworks tend to be more complex than traditional software in terms of design elements, as AI often involves complex mathematical models and algorithms [8, 110, 56, 111]. In the past years, there have been multiple research studies to characterize AI bugs which are in various stages of the AI development pipeline, including data preprocessing, feature engineering, model training, and inference [112, 56, 113, 114], or implementation bugs which are in the code of the AI frameworks, such as bugs in the core algorithms or components of the library [111, 110, 115, 114, 116]. For example, Zhang et al. [113] studied the bugs and issues that developers encounter when working with TensorFlow, a popular open-source DL library developed by Google. Islam et al. [116] conducted the first study on characterizing API usage bugs of five DL libraries, including Caffe, Keras, TensorFlow, Theano, and Torch. They provided a classification for bug types, root causes, impact, and the DL development stage where bugs occur. Despite these efforts, the characteristics of implementation bugs in AI frameworks have not been well studied, which leaves unanswered the more directly relevant questions: *What kinds of bugs are found in the studied AI frameworks? What are the root causes of bugs in the studied AI frameworks? What symptoms do these bugs have? and Are there any fixing patterns for resolving these bugs?*

Understanding such characteristics of bugs in AI frameworks has the potential to foster the development of secure and reliable AI platforms. To fill the above research gap, we take the first step towards characterizing and understanding bugs in AI frameworks. More specifically, we conduct the first comprehensive study to explore four significant factors: 1) bug types, 2) root causes, 3) symptoms, and 4) fixing patterns of bugs in seven well-known AI frameworks including TensorFlow [117], PyTorch [72], Scikit-learn [75], Mlpack [118], Pandas [119], Numpy [120], and Scipy [121].

In this chapter, we considered all available commits in the default branch of the Github repositories of the studied AI frameworks when we conducted this study on Sept. 1st, 2021, to collect bugs. We first searched commit titles and messages with bug-related keywords (details are in Section 2.2.1) to identify bug-fixing commits. As a result, more than 5K commits are collected. We then manually checked each

commit collected in the first step and identified and characterized bugs from them by following systematic processes (details are in Section 2.2.2). In total, we obtained 683 unique bugs from the subject AI frameworks. In this chapter, we are to address the following research questions:

- **RQ1:** What types of bugs exist in the studied AI frameworks?
- **RQ2:** What are the root causes for bugs in the studied AI frameworks?
- **RQ3:** What are the symptoms of bugs in the studied AI frameworks?
- **RQ4:** What are the fixing patterns for bugs in the studied AI frameworks?

This chapter makes the following contributions:

- To the best of our knowledge, we conduct the first empirical study to characterize and understand bugs in AI frameworks.
- We show the detailed taxonomies regarding the types, root causes, symptoms, and fixing patterns of bugs in AI frameworks.
- We provide a set of practical guidelines to help machine learning development teams develop reliable and secure AI frameworks.
- We release the dataset and source code of our experiments to help other researchers replicate and extend our study.

The rest of this chapter is organized as follows. Section 2.2 presents the methodology of this study. Section 2.3 presents the results. Section 2.4 shows the threats to the validity of this work. Section 2.5 presents the related studies. Section 2.6 concludes this chapter.

2.2 Methodology

2.2.1 Data Collection

Library Selection: We design the following inclusion criteria for the selection of the studied AI frameworks from GitHub: 1. Accessibility, i.e., the libraries should be popular, widely used, and open-source, 2. Maturity, i.e., the libraries should have been actively developed for a considerable amount of time, 3. Comprehensiveness, i.e., the libraries cover the current industrial AI practice and represent the critical

aspects of AI application developments, 4. Applicability, i.e., the libraries should support classical and state-of-the-art AI models, data processing and manipulation, statistical models, etc., and 5. Availability, i.e., the libraries should have sufficient bug-fixing commits stored in their GitHub repositories. Initially, we selected the following libraries, i.e., Mlpack, TensorFlow, PyTorch, Numpy, Pandas, Scikit-learn, Pandas, Theano, Keras, Matplotlib, and Caffe. While Theano, Keras, and Caffe do not have sufficient bug-fixing commits in their GitHub repositories. Matplotlib does not meet the applicability criteria. Eventually, using the criteria, we select seven AI frameworks including TensorFlow [117], PyTorch [72], Scikit-learn [75], Mlpack [118], Pandas [119], Numpy [120], and Scipy [121] as our experiment subjects.

Table 2.1: Statistics of subject AI frameworks used in this chapter.

AI frameworks	#CVEs	#Commits	# Bugs	Language
Tensorflow	36	1,197	250	C++/Python
PyTorch	N.A	563	75	C++/Python
Sickit-Learn	N.A	325	37	Python
Pandas	N.A	869	84	Python
Mlpack	N.A	664	46	C++
Numpy	N.A	1,198	149	C/Python
Scipy	N.A	793	42	C/Python
Overall	36	5,609	683	-

Bug Fixing Commit Collection: Our automatic fixing commit collection procedure is based on the rule heuristics implemented based on a set of bug-related keywords extracted from [122]. The outcome of automatic filtering is 5,609 commits that are identified as bug candidate commits. Note that such an automatic approach introduces false positives since the employed regex patterns are either too broad or not specific enough, leading to matches with commits that are not bug-related. Hence we further started a manual labeling process to identify bug-related commits.

Bug-fixing Commit Identification: This subsection explains the adopted steps to filter out noises and identify bug-related commits from the 5,609 commits extracted in the initial phase. We adopt the following steps:

Check Possible CVE Numbers: If there is any CVE number associated with a commit, we consider the commit as a bug-related commit as CVE records are verified and confirmed by experts and researchers to track, analyze, and report on software bugs.

Analyze Commit Message and Title: In this step, we look for specific bug-related keywords in the title and the message of commits, e.g., *buffer overflow*, *stack overflow*, *RCE*, *exposure*, and *memory leaks*, etc.

Analyze Code Changes: We review the code changes associated with each commit to see if they address bug-related commits. For example, we check changes that involve adding or removing null checkers to determine whether the corresponding commit is related to fixing a null pointer dereference.

Analyze Linked Issues and Pull Requests: When a bug is discovered, it is typically reported through an issue on the project’s issue tracker, which may include details about the bug, such as how it can be exploited and what the potential impact is. Consequently, we review these linked issues to determine why a particular code change was made and whether it was made in response to a known issue.

2.2.2 Data Labeling

In our study, we analyze each bug-fixing commit from multiple aspects: 1) bug type, 2) root cause, 3) symptom, and 4) fixing pattern. Please note that some existing studies on analyzing general software bugs in AI frameworks have also provided taxonomies for these aspects [56, 114]. In this work, we did not adopt corresponding taxonomies from these studies. The reason is that existing studies mainly focus on the characteristics of general software bugs in AI frameworks. As a result, it is not valid to adopt their classifications since general software bugs’ characteristics can be significantly different. To build bug taxonomies for the studied AI frameworks, we incrementally created them as we analyze and review each bug-fixing commit [123, 124, 125]. For each commit, the first two authors analyze the commit data based on the defined research questions and take into account the following steps:

Analyze the Bug-Fixing Commits: We analyze the Bug-Fixing commits data (commit title, message, code change, discussions, issues, and pull requests) to identify if there exist patterns and characteristics that are common among the bugs (details are in Section 2.2.1). We consider all 683 commits in the first round of reviewing.

Create Preliminary Taxonomies: Based on the analysis in the previous step, we create a preliminary taxonomy that categorizes the bug-fixing commits based on their types, root causes, symptoms, and fixing patterns.

Expand Taxonomies: We continue the previous step and assign new commits to the preliminary taxonomy. If we find a new commit that cannot fit into the preliminary taxonomy regarding its type, root cause, symptom, or fixing patterns, we expand the preliminary taxonomy by adding a new type created from the commit. Note that, the authors worked together to decide on all the disagreements during the above processes.

2.3 Result Analysis

2.3.1 RQ1: Bug Types

We organized bug types into five high-level categories (shown in Figure 2.3) (i.e., *Memory*, *Numeric*, *Buffer*, *Resource*, *Concurrency*), involves more than 16 different Common Weakness Enumeration (CWEs)⁵ covered by 621 (90.9%) of the 683 bugs. The remaining 62 (9%) bugs that appear infrequently and do not belong to any particular groups are included in the *Others* category.

Memory: Memory bugs happen when AI framework developers improperly handle memory allocation and deallocation. This category accounts for 226 (33%) of the bugs. Specifically, it contains the following five types of CWEs: 1) *Missing Release of Memory after Effective Lifetime (Memory Leak (CWE-401)*, 2) *Null Pointer Dereference (CWE-476)*, 3) *Infinite Loop (CWE-835)*, 4) *Double Free (CWE-415)*, and 5) *Use After Free (CWE-416)*.

Numeric: AI frameworks perform a large number of tensor-level computations, more specifically dealing with floating point operations, which results in *Numeric* bugs. This category accounts for 198 (28.9%) of the bugs. It mainly has four types of CWEs: 1) *Integer Overflow (CWE-190)*, 2) *Insufficient Precision or Accuracy of a Real Number (CWE-1339)*, 3) *Division by Zero (CWE-369)*, and 4) *Integer Underflow (CWE-191)*.

Buffer: Bugs related to buffer also known as buffer overflows or overruns, occur when a computer writes more data to a temporary storage space in memory known as a buffer than it can manage. This category accounts for 96 (14%) of the bugs. It mainly covers five types of CWEs: 1) *Out of Bound Read (CWE-125)*, 2) *Stack Overflow (CWE-121)*, 3) *Heap Buffer Overflow (CWE-122)*, 4) *Buffer Overflow (CWE-120)*, and 5) *Out of Bound Write (CWE-787)*.

Resource: Resource bugs occur when a program fails to manage resources such as memory, and file handles. This category accounts for 54 (7.9%) of the bugs. It consists of only one subcategory: *Use of Uninitialized Resource (CWE-908)*

Concurrency: Concurrency in AI frameworks is a type of software weakness that occurs due to their complexity, a large number of computations that are not thread-safe, and asynchronous execution of computation graphs. This category accounts for 47 (6.8%) bugs. It consists of two types of CWEs: 1) *Race Condition (CWE-362)* and 2) *Deadlock (CWE-833)*.

⁵CWE (<https://cwe.mitre.org/>) is a well-known and widely accepted standard for documenting and categorizing bugs. The use of CWE categories can assist in standardizing bug descriptions and make it easier for researchers and practitioners to comprehend and explain the nature of issues:

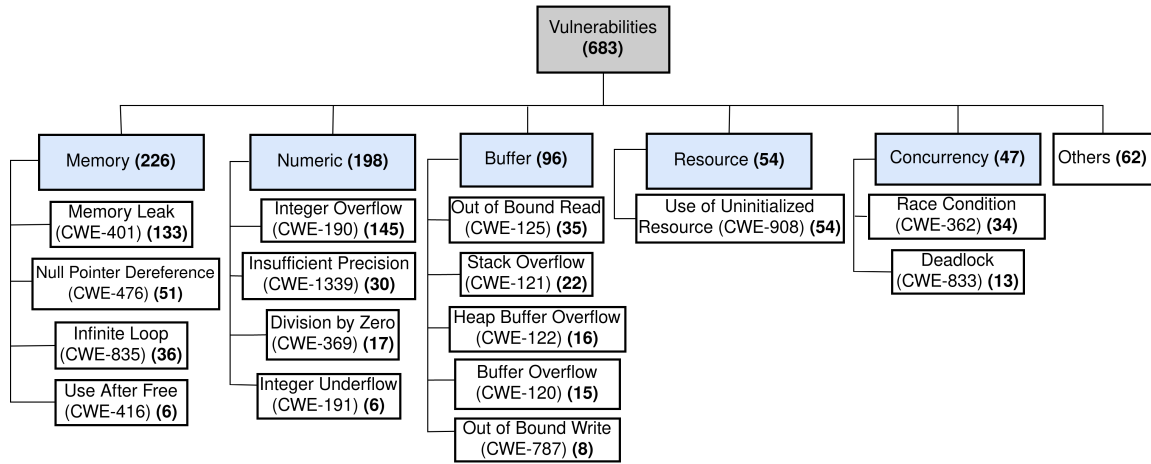


Figure 2.3: The taxonomy of bug types studied in this work.

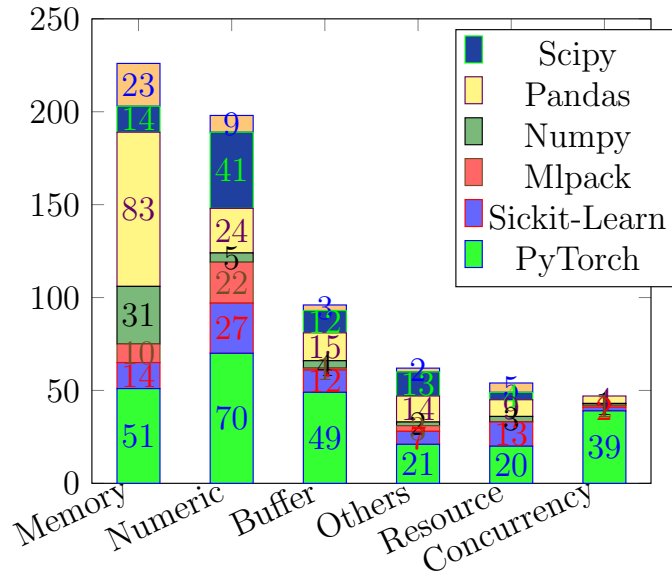


Figure 2.4: The distribution of bugs in the subject AI frameworks.

Implication: AI frameworks use large tensors to represent data, which can consume a lot of memory. These intensive usages of tensors require frequent allocation and deallocation of memory in the implementation of AI frameworks (memory management) leading to bugs such as buffer overflow and memory leaks. Buffer

overflows occur when data is written to a buffer more than it can contain, and can cause data to overflow into neighboring memory areas, potentially overwriting important data or executing malicious code. As shown in Table 2.2, *Memory Leak* and *Null Pointer Dereference* are the two most common subcategories of *Memory* bugs. *Memory Leak* is common in the Numpy library where memory management should be done manually due to the development of language barriers. To make the matter more concrete, we elaborate on an example of *Memory Leak*⁶ where the developer has created a slice object but never released by *decref* statement. According to the Python C API reference documentations⁷, if a created object is going to be returned by the function in which it is created, the reference to the object should be decremented.

AI frameworks frequently perform calculations on the size of arrays, multiplication of tensors together, or multiplying or adding integer values. For example, AI frameworks may use a 32-bit integer type to store the result of a multiplication operation that produces a value greater than $2^{31} - 1$ (the maximum value that can be represented by a signed 32-bit integer), an integer overflow will occur and the value will wrap around to a negative value. This can result in numerical instability such as leading to poor model performance or crash. Attackers can exploit the ranges of the defined variables and cause denial of service attacks or make the result of model training and inference inconsistent. An example of integer overflow bug found in the Tensorflow library⁸ where the developer has defined integer variables with 32 bits precision which cause overflow. The suggested fix is defining integer variables with higher precision, e.g., in this commit, the author has defined 64-bit integer variables.

Comparison with Traditional Software: We further check whether there is a similar trend regarding the distribution of bug types in traditional software systems or not. We based our comparison according to the well-known paper published on characteristics of traditional software systems [126]. According to Tan et al. [126], *Memory Leak (CWE-401)* is the most frequent bug type which is similar to our finding and confirms that *Memory Leak (CWE-401)* is the dominant bug type in both the studied AI frameworks and general software systems. We also find that *Integer Overflow (CWE-190)* is the dominant bug type in the category of *Numeric*. This trend is not similar to general software systems [126] and further confirms our findings that *Numeric* is significant to AI frameworks as they rely on heavy computations using tensors and arrays, based on floating point operations. We further check a recently published paper [127] which characterizes bugs in web assembly compilers.

⁶https://github.com/numpy/numpy/commit/4e19f408de900f958441af4ec8a45_8f5ce6473eb

⁷<https://docs.python.org/3/c-api/memory.html>

⁸<https://github.com/tensorflow/tensorflow/commit/087859fce9409991164f727735743da4cb310fd4>

Table 2.2: Subcategories of Memory. ML denotes Memory Leak, NPD is Null Pointer Dereference, IL is Infinite Loop, and UAF is Use After Free.

Library	ML	NPD	IL	UAF	Overall
Tensorflow	6	25	19	1	51
PyTorch	2	7	5	0	14
Sickit-Learn	8	0	2	0	10
Mlpack	26	3	2	0	31
Pandas	5	6	3	0	14
Numpy	64	10	5	4	83
Scipy	22	0	0	1	23

Table 2.3: Subcategories of Numeric. IO is Integer Overflow, IP is Insufficient Precision, DZ is Division By Zero, and IU is Integer Underflow.

Library	IO	IP	DZ	IU	Overall
Tensorflow	63	4	2	1	70
Pytorch	18	6	3	0	27
Sickit-Learn	8	4	7	3	22
Mlpack	2	2	1	0	5
Pandas	34	3	3	1	41
Numpy	13	10	1	0	24
Scipy	7	1	0	1	9

They find that *Data type incompatibility* accounts for 15.75% of bugs in web assembly compilers. These bugs arise from incompatibility in interfaces between WebAssembly and JavaScript. This pattern is not similar to *Numeric* bugs in the studied AI frameworks as they have very sophisticated bug patterns.

Finding 1: *Memory* and *Numeric* are the dominant categories of bug types across AI frameworks, accounting for 33% and 28.9% of bugs

2.3.2 RQ2: Root Causes

Root causes in the subject AI frameworks are organized into five high-level categories (illustrated in Figure 2.6) including *Data Type*, *Memory*, *API*, *Business Logic*, and *Concurrency* covered by 619 (90.6%) of the 683 bugs. The remaining 64 (9.3%) root

Table 2.4: Subcategories of Data Type Errors. LVTAP is Lack of Validating Tensors and Arrays Property, IVRI is Integer Variable Range Issue, UIDT is Using Improper Data Type, LOC is Lack of Overflow Checking, FVRI is Float Variable Range Issue, and ONT is Out of Bound Nanosecond for Timestamp.

Library	LVTAP	IVRI	UIDT	LOC	FVRI	ONT	Overall
Tensorflow	42	21	20	7	4	0	94
Pytorch	14	8	8	4	4	0	38
Sickit-Learn	0	3	1	3	1	0	8
Mlpack	2	0	2	0	1	0	5
Pandas	1	8	8	7	1	9	34
Numpy	14	4	5	5	0	0	28
Scipy	0	0	5	3	0	0	8

causes have no clear indication about their types, and hence we group them in *Others* category.

Note that although some of the high-level taxonomy names from bug types (RQ1) and root causes (RQ2) are similar, they refer to different levels of abstraction in Figure 2.3 (RQ1) and Figure 2.6 (RQ2). For example, In RQ1, the *Memory* category refers to a broader range of memory bugs. In RQ2, *Memory* is a more specific category that explains the root cause of bugs depicted in Figure 2.3.

Data Type: This root cause category accounts for 215 (31.4%) of bugs. Subcategories include 1) *Lack of Validating Tensors and Arrays Property*: Sometimes implementations fail to check tensors or array properties, e.g. shape or rank. As a result, attackers can exploit tensor or arrays properties and craft special inputs to trigger denial of service via segmentation fault or overflows, 2) *Using Improper Data Type*: When developers have confusion about what kind of data types should they use for variables, e.g. using a *float* instead of a *double*, 3) *Integer Variable Range Issue*: When developers define integer variables with limited range or very large range, e.g. defining *int32* bits instead of *int64* bits, 4) *Lack of Overflow Checking*: If the program does not check for overflow, the result of the operation may be incorrect, which can lead to bugs in AI frameworks, 5) *Float Variable Range Issue*: Similar to *Integer Variable Range Issue*, for example using *float32* instead of using *float64* data type for numeric operations, 6) *Out of Bound Nanoseconds for Timestamp*: Generally, pandas timestamp data type represents date object in nanosecond resolution which failure in storing date in its default range results in out of bound bug.

Memory: This root cause category accounts for 30.4% of bugs. The subcategories are including 1) *Improper Memory Management*: When a developer has confusion in memory management, either misuses a memory release statement or forgets to release

memory after its effective lifetime, 2) *Invalid Memory Access*: When a process tries to access memory locations filled with null values, corrupted, already been deleted, or freed, 3) *Very Deep Tensors and Computation Graphs*: Sometimes tensors or computation graphs grow unexpectedly at runtime and exceed stack or buffer size which results in a segmentation fault, 4) *Uninitialized Resource*: An uninitialized resource bug occurs when a program or application fails to properly initialize a resource, such as a variable or a file, before using it. This can lead to unexpected behavior and potentially allow an attacker to exploit the bug and execute malicious code or steal sensitive information.

Business Logic: This category of the root causes accounts for 95 (13.9%) of bugs. It includes 1) *Improper Exception Handling*: When developers incorrectly handle exceptional conditions leading to termination of the software during normal executions of the software, 2) *Incorrect Index Calculation*: Normally, tensors or arrays are accessed by indices, however, sometimes indices are vulnerable to many attacks, e.g. assigning a large value to an index, which cause array out of bound access bug and trigger a denial of service via crash or segmentation fault, 3) *Wrong Order of Execution*: When the execution order of instructions in the backend implementation is not in the intended order. This can lead to unexpected behavior, data corruption, or crashes.

API: This category accounts for 71 (10.3%) of total records. Subcategories include 1) *API Misuse*: When developers mistakenly use a specific API, e.g., passing parameters in wrong orders, lack of using optional parameters, and mistakenly using optional parameters, 2) *Using Wrong API*: When developers mistakenly use improper APIs, e.g. using `numpy.empty()` instead of using `numpy.empty_like()`, 3) *Malicious Parameters*: When developers pass malicious or invalid parameters to API calls that are exploitable by attackers. Attackers can exploit these parameters by crafting particular inputs to take control of AI frameworks, and 4) *Third Party Library Issue*: When developers mistakenly use either wrong versions of APIs or outdated ones.

Concurrency: This category involves concurrent access of resources in a shared environment by multiple threads due to improper resource locking, releasing, or simultaneous resource access accounting for 30 (4.3%) bugs. Subcategories are 1) *Missing Locking Statement*: When developers forget to lock resources which mostly results in race condition or deadlock bugs, 2) *Improper Usage of Locking Statement*: When developers use locking statements improperly on resources, or they release locked resources inappropriately, which may result in deadlock or race condition bugs, and 3) *Improper Resource Locking*.

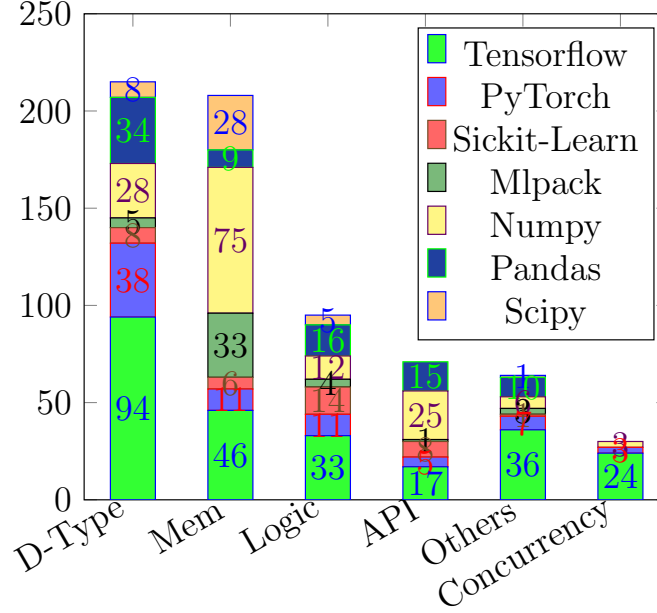


Figure 2.5: Distribution of root causes across AI frameworks.

Figure 2.5 shows the distribution of root causes of bugs in different libraries. As can be seen, the *Data Type* category is the most common root cause of bugs across the subject AI frameworks. *Memory* is the second most common root cause of bugs across the subject AI frameworks. Table 2.4 and Table 2.5 further show the distribution of subcategories of *Data Type* and *Memory* across the subject AI frameworks respectively. As we can see, *Lack of Validating Tensors or Arrays Property* and *Integer Variable Range Issue* are the major subcategories in *Data Type*. While *Improper Memory Management* and *Invalid Memory Access* are the dominating subcategories in *Memory*.

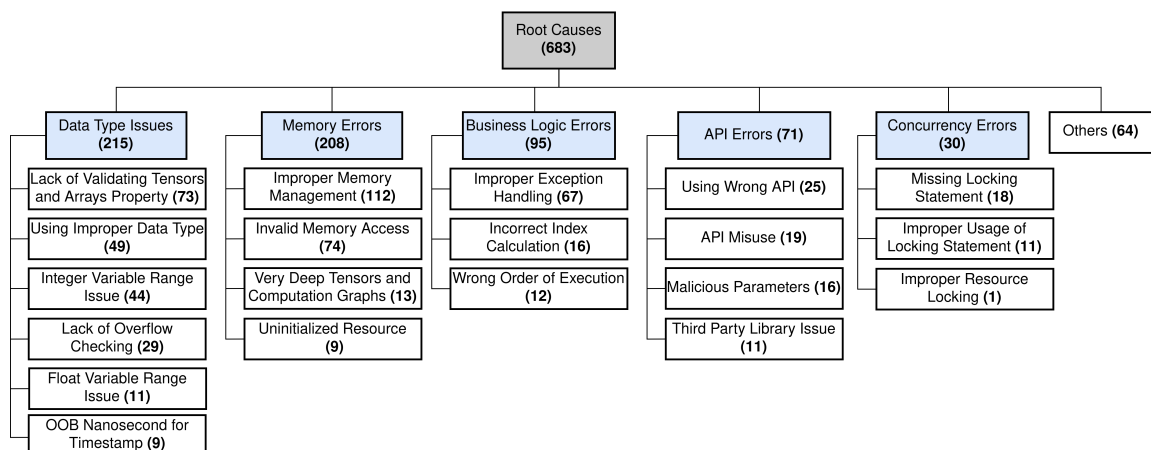


Figure 2.6: Taxonomy of root causes in AI frameworks.

Implication: Tensors and arrays have properties such as shape, rank, axis or dimensions, and size which can be vulnerable to attacks from external attackers. These bugs can be exploited by passing invalid values, resulting in out-of-bounds read and denial of service through a segmentation fault or crash. An intuitive example of such attacks is this bug from Tensorflow library⁹ where *tf.raw_ops.QuantizeAndDequantizeV2* allows the *axis_* argument to accept invalid arguments. In this example, the constraint is that *axis_* accepts -1 and the checker (*OP_REQUIRES*) is true as long as the value is less than or equal to -1, though it results in heap underflow bug which is exploitable by attackers to write or read their data on top of the heap.

AI framework developers use statistically typed languages for development which can be confusing for developers, as they must decide the range of integer or float variables. A very high precision range can result in performance degradation and resource consumption, while a low precision range can cause loss and unexpected behavior in the AI frameworks. These difficulties can lead to mistakes by developers and may lead to defining variables with improper range precision. For example, the

⁹<https://github.com/tensorflow/tensorflow/commit/c5b0d5f8ac19888e46ca14b0e27562e7fbbee9a9>

Table 2.5: Subcategories of Memory Errors. IMM is Improper Memory Management, IMA is Invalid Memory Access, and VDTC is Very Deep Tensors and Computation Graphs, and UR is an Uninitialized Resource.

Library	IMM	IMA	VDTC	UR	Overall
Tensorflow	3	35	8	0	46
Pytorch	1	9	1	0	11
Sickit-Learn	6	0	0	0	6
Mlpack	21	5	0	7	33
Pandas	3	5	1	0	9
Numpy	57	13	3	2	75
Scipy	21	7	0	0	28

default integer variable in the Tensorflow library has 32 bits, we observed that defining *int32* bits is one of the major root causes of bugs. For example, in this commit from Tensorflow library¹⁰, the developer has defined *output_elements* as a *const int32* bits. The cause *CalculateTensorElementCount()* to assign a very large value in a 32-bit integer variable and cause integer overflow. The suggested fix is using *int64* range precision to prevent the bug.

Finding 2: *Data Type* category is the most common root cause accounting for 31.4% bugs in the subject AI frameworks. This is because developers are confused about the range of integer or float variables during the development process. High precision ranges can lead to performance degradation and resource consumption, while low precision ranges can cause loss and unexpected behavior in the model.

Comparison with Traditional Software: We start by investigating the difference between AI frameworks versus traditional software in terms of *Data Type* and *API* as two important root causes of bugs in the subject AI frameworks.

Similar to the findings in *RQ1*, we find that *Data Type* category is the dominant category of root causes in the subject AI frameworks, with *Lack of Validating Tensors and Arrays Property* as the most frequent subcategory. Due to this dominance, we further check whether root causes in traditional software systems follow the same pattern or not. Consequently, we compare our findings with the findings in [126] and [128]. Hirsch and Hofer [128] analyze the root cause of bug reports gathered from 103 Github projects. According to our investigations, we find that in both papers, semantic bugs are the major dominant root cause of bugs accounting for 87% (maximum percentage which belongs to Mozilla project¹¹) and 269/512 (52.5%) in

¹⁰<https://github.com/tensorflow/tensorflow/commit/087859fce9409991164f727735743da4cb310fd4>

¹¹In [126], semantic bugs account for 82.5% and 70.1% in Apache and Linux projects, respectively.

[128] and [126]. As a result, the root cause patterns in traditional software are not similar to patterns of root causes in the subject AI frameworks.

API bugs in the subject AI frameworks and general software are distinct in terms of their nature and impact. One distinction is that AI frameworks are frequently used for more sophisticated tasks like image or speech recognition, which necessitate a greater level of precision and accuracy. Additionally, AI frameworks often use third-party dependencies (such as numerical libraries such as FP16, absl, fft2d in TensorFlow or BLAST, and LAPACK in Numpy) to perform numerical computations. If these dependencies are not properly maintained or updated, it can cause bugs in the AI framework. For example, if a bug is discovered in a third-party library used by an AI framework, it can cause the AI framework to be vulnerable to attacks as well. Overall, API bugs in AI frameworks are more complicated to solve than in traditional software, although the development process and debugging methodologies are similar.

We further investigate the difference between API bugs in the subject AI frameworks versus traditional software highlighted by other researchers. According to a current study on *API* bugs conducted by Amann et al., [129], missing and redundant API calls are the most frequent bugs. At the same time, these are the least bugs in the subject AI frameworks. We conclude that developers of the subject AI frameworks have difficulty understanding which APIs they should use, how to use them, and make them secure.

Finding 3: *Lack of Validating Tensors and Arrays Property* is the most frequent root cause of bugs in the subject AI frameworks. If the properties of tensors and arrays are not properly validated, it can lead to a crash, segmentation fault, or unexpected behavior in the model. This can happen because of a lack of proper validation checks in the library’s backend implementation.

Finding 4: *API* in the subject AI frameworks have distinct bug patterns compared to traditional software which makes testing and debugging them difficult and time-consuming.

2.3.3 RQ3: Symptoms

Symptoms of bugs in the subject AI frameworks are organized into 4 categories including *Segmentation Fault*, *Crash*, and *Unexpected Behaviour*, *Resource Consumption*, covered by 677 (99.1%) bugs. The remaining 6 bugs have no clear indication about their outcome, and hence we group them in *Others* category.

Table 2.6: Subcategories of API Errors. AM is API Misuse, MP is Malicious Parameters, TPLI is Third Party Library Issue and UWA is Using the Wrong API.

Library	AM	MP	TPLI	UWA	Overall
Tensorflow	2	2	6	7	17
Pytorch	1	1	1	2	5
Sickit-Learn	3	2	0	3	8
Pandas	4	7	1	3	15
Numpy	9	4	2	10	25
Mlpack	0	0	1	0	1
Scipy	0	0	0	0	0

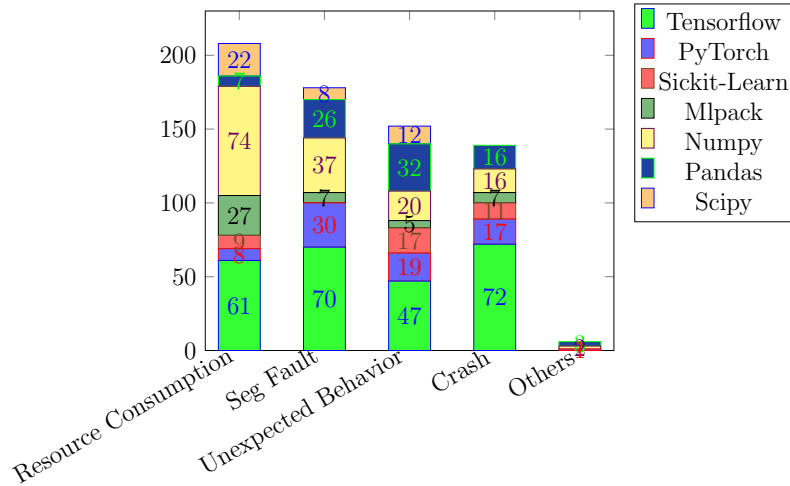


Figure 2.7: Distribution of symptoms across AI frameworks.

Resource Consumption: Accounting for 30.45% (208) of bugs, is the exhaustion of available resources, e.g., increasing memory usage because of uncontrolled or improper allocation of the memory.

Segmentation Fault: Accounting for 26% (178) of bugs, when a program outputs *Segmentation fault* or *Segmentation fault (core dumped)* in the output which is the symptom for segmentation fault¹².

Unexpected Behavior: Accounting for 22.2% (152) of bugs, happens when the library produces results that are not expected. For example, in this *Integer Overflow*

¹²<https://stackoverflow.com/questions/49092527/illegal-instructioncore-dumped-tensorflow>

bug from Scikit-learn library¹³ where $pk * qk$ returns *inf* instead of *float* because of exceeding int32 bits limits during multiplication.

Crash: Accounting for 20.3% (139) of bugs, happens when a program or application terminates unexpectedly, is called *Crash* also known as *crashing*. This can happen for a variety of reasons, such as accessing an invalid memory location, a divide-by-zero bug, or an infinite loop.

Figure 2.7 demonstrates the distribution of symptoms across different libraries. As you can see, the most frequent symptom is *Resource Consumption* accounting for 30.45% of bugs. We also draw a mapping from root causes to symptoms to interpret what is the outcome of bugs as shown in Figure 2.8. It is observable from Figure 2.8 that the majority of bugs caused by *Memory* have *Resource Consumption* as their impact. Also, the majority of bugs caused by *Data Type* have *Unexpected Behavior* as their impact. One possible usage scenario of bug symptoms in the subject AI frameworks is that AI developers can narrow down the types of bugs present in their codebase and take steps to fix them. Also, AI developers do not need to develop test oracles for unit testing of the subject AI frameworks to understand bugs that have *Segmentation Fault* and *Crash* as their symptoms.

Finding 5: *Resource Consumption* and *Segmentation Fault* are the most common symptoms of bugs accounting for 30.45% and 26% of bugs respectively. The subject AI frameworks are vulnerable to resource consumption bugs because they are designed to perform computationally intensive operations on large amounts of data. These operations can consume a significant amount of memory and processing power, making it easy for bugs to cause the program to consume too many resources and crash or become unresponsive.

2.3.4 Comparison with Traditional Software

We find that *Resource consumption* and *Segmentation Fault* are two dominant symptoms of bugs in the subject AI frameworks. There is still an important question that needs to be addressed: *What is the difference between symptoms in traditional software systems and AI frameworks?* To answer this question, we compare our findings with the study conducted by Tan et al. [126]. According to the findings in [126], *Functional* impact is the major symptom of bugs in traditional projects which greatly deals with the fact that the software system is not functioning according to its specifications. However, regarding AI frameworks, we have a completely different

¹³<https://github.com/scikit-learn/Scikit-learn/commit/622f912095308733ddfe572a619b1574b9da335e>

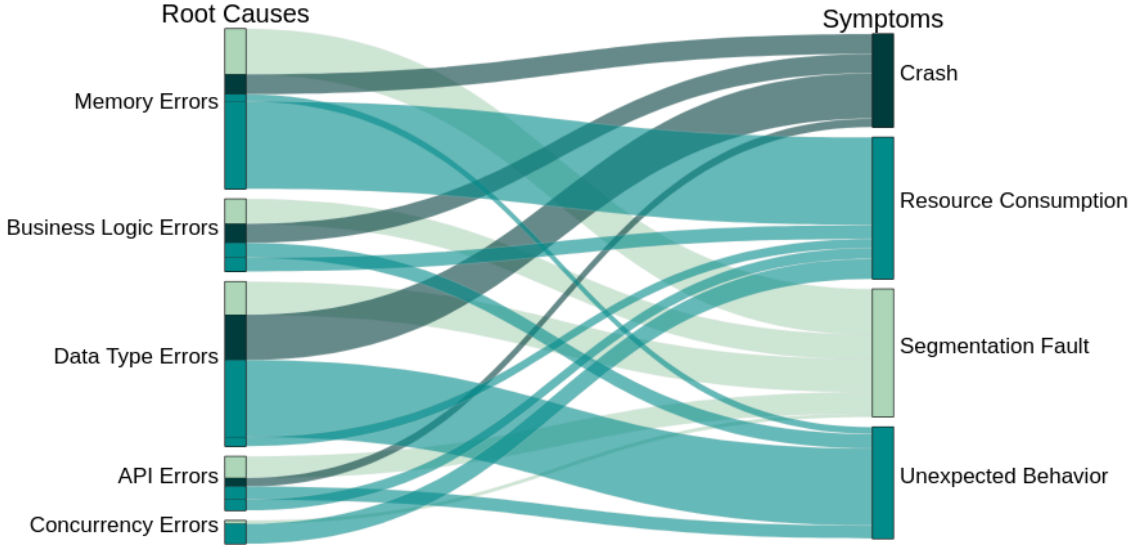


Figure 2.8: Mapping of root causes to symptoms.

trend where the majority of symptoms of bugs in the subject AI frameworks result in the crashing of the library during runtime via *Segmentation fault* which is the dominant symptom.

2.3.5 RQ4: Fixing Patterns

Fixing patterns are organized into six high-level categories (shown in Figure 2.9) including *Add Checkers*, *Resolve Data Type bugs*, *Resolve Memory bugs*, *Resolve API bugs*, *Resolve Concurrency bugs*, and *Modify Business Logic Errors* covered by 586 (85.7%) of bugs. The remaining 97 (14.2%) fixing patterns have no clear indication about their types and hence are included in the *Others* category.

Add Checkers: Fixing patterns in this category mainly relate to the addition of either library-specific or conventional checkers to fix bugs, accounting for 155 (22.6 %) bugs. Subcategories are 1) *Add Checker for Tensors and Arrays Property*: This is the most common fixing pattern where developers use if conditions or library-specific checkers to validate tensor and arrays properties, e.g., shapes, ranks, values, or elements, 2) *Add Checker for Null Pointer Dereference*: Developers often add checkers either using if conditions or library-specific checkers to prevent null pointer dereferences, 3) *Add Checker for Overflow*: This fixing pattern is mainly used to fix overflow bugs where developers either add if modules or library-specific checkers to prevent overflow or throw appropriate exceptions.

Resolve Memory Bugs: Fixing patterns in this category relates to memory management efforts, which fix 130 (19%) of total bugs. Subcategories are 1) *Manage Memory Release*: is used to fix bugs related to incorrect or inappropriate memory allocations and 2) *Resource Initialization*: when developers initialize tensors or variables to fix bugs.

Modify Business Logic: It accounts for 16.8% of bugs (115), related to improving exception handling, file handling, control flows, methods, or classes having incorrect logic. Subcategories are 1) *Improved Exception Handling*: When a program crashes, it is necessary to raise appropriate message to help developers with debugging operations. This pattern adds missing error reporting or modifying existing exception handling, 2) *Modify Index Calculation*: This pattern is used to fix bugs caused by improper calculation of tensors and arrays indices, 3) *Avoid Overflow on Deep Tensors and Graphs*: This pattern is used when developers try to prevent overflow caused by small stack size or deep computation graphs created in runtime, 4) *Modify Order of Execution*: Developers change the location of semantically related statements to fix bugs, 5) *Close File Handler*: It is used to fix file descriptor leak bug where developers sometimes forget to close opened files after its lifetime.

Resolve Data Type Bugs: Fixing patterns in this category focus on resolving bugs related to data types, which cover 14.7% (101) of bugs. Subcategories include 1) *Adjust Integer Type Range*: This pattern is used to either increase or decrease the range precision of integer variables to prevent numerical overflow bugs, 2) *Modify Data Type*: When data types are defined incorrectly, this pattern alter existing defined data types to fix corresponding bugs, 3) *Adjust Float Type Range*: Similar to *Adjust Integer Type Range*, this pattern adjusts float variables range precision, e.g., either increase the range precision or decrease it.

Resolve API Bugs: Fixing patterns in this category are mainly used to fix bugs introduced by inappropriate API usages, which help fix 60 (8.7%) of bugs studied in this chapter. The detailed subcategories are 1) *Using Proper API*, 2) *Update API Usage*, and 3) *Update Third Party Library*.

Resolve Concurrency Bugs: Accounting for 25 (3.6%) bugs, this category of fixing pattern is used to fix bugs related to concurrency issues resulting in deadlock or race condition errors. Subcategories include 1) *Add Locking Statement*, 2) *Modify Locking Statement*, and 3) *Remove Locking Mechanism*.

2.3.6 Implication

We find that developers of the studied AI frameworks frequently use validation checks in the code, such as checking the shape and data type of tensors and arrays before

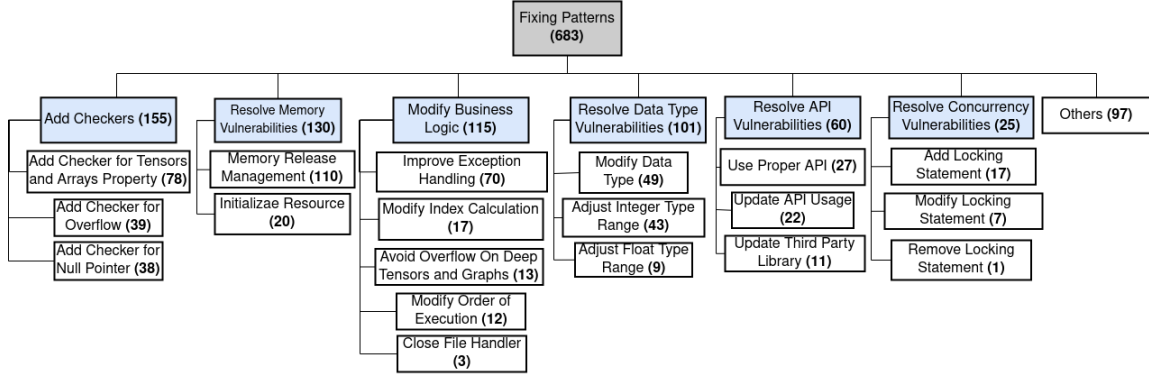


Figure 2.9: Taxonomy of fixing patterns in AI frameworks.

using them in computations. In this example¹⁴ which is *Integer Overflow*, there is no checker on `data[axis]` to make sure it does not exceed int32 bits range limits. The developer overcomes the problem by adding a checker of `TF_LITE_ENSURE` on line 76 in `tensorflow/lite/kernels/concatenation.cc`. We also find that developers of the studied AI frameworks usually mitigate *Data Type Bugs* by explicitly setting data types when creating tensors, for example, using `tf.float32` instead of `tf.float` for a tensor.

Finding 6: *Adding Checkers for Tensors and Array Properties* is the most common fixing pattern used by developers accounting for 11.4% of bugs. In this pattern, developers often implement solutions such as adding validation checks either using library-specific macro checkers or general if statements to guard against weaknesses corresponding to lack of validation:

¹⁴<https://github.com/tensorflow/tensorflow/commit/4253f96a58486ffe84b61c0415bb234a4632ee73>

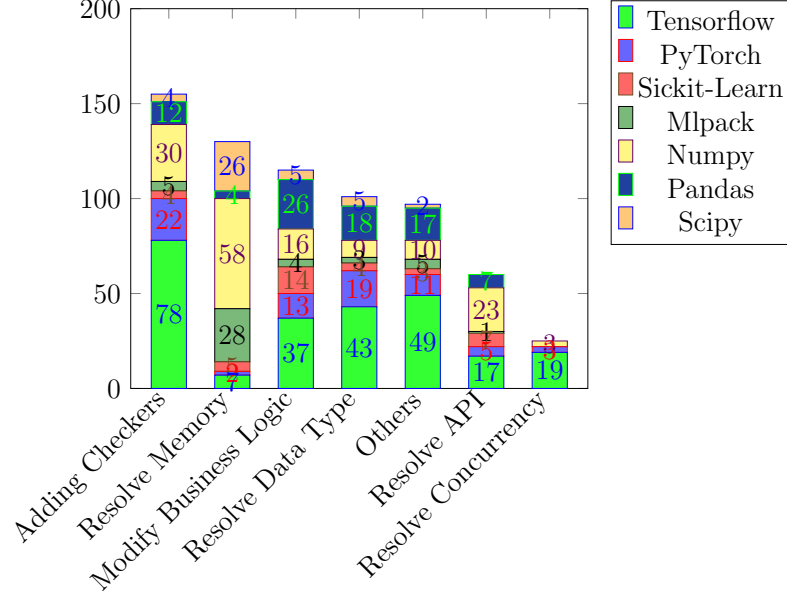


Figure 2.10: Distribution of fixing patterns across different AI frameworks.

Table 2.7: Subcategories of Add Checkers. ACTAP is Add Checker for Tensors and Arrays Property, ACO is Add Checker for Overflow, and ACNP is Add Checker for Null Pointer Dereference.

Library	ACTAP	ACO	ACNP	Overall
Tensorflow	45	14	19	78
Pytorch	14	5	3	22
Sickit-Learn	0	3	1	4
Mlpack	2	0	3	5
Pandas	2	9	1	12
Numpy	14	5	11	30
Scipy	1	3	0	4

2.4 Threats to Validity

Construct Validity: Our study is primarily aimed at identifying and examining bugs present in AI frameworks, rather than evaluating their severity. Although CVSS

severity data can provide important information about bugs, it is important to note that other factors can also contribute to their impact. Therefore, our focus is on identifying and classifying bugs based on their types, root causes, symptoms, and fixing approaches as they relate to the functionality of the AI frameworks, rather than assigning severity scores.

Internal Validity: The main internal threat to our work is our manual analysis labeling and classification of bugs which may suffer from subjective bias and errors. To guard against this, the first authors who are skilled in bugs have reviewed the collected commits. Authors also discuss any possible disagreement until a consensus is reached.

External Validity: The dominant threat to the external validity of this study is the collected dataset. To overcome this threat, we collected commits from seven different AI frameworks; two are very famous and widely used DL libraries, including TensorFlow and PyTorch; two of them are Mlpack and Scikit-learn which are renowned classical AI frameworks which often is used beside DL libraries. We also collected data from three well-known data analytical and visualization tools, including Pandas, Numpy, and Scipy. The reason behind this diverse data collection is to generalize our findings to wide domains and increase the reliability of findings.

2.5 Related Work

2.5.1 Studies on General Bugs

There are many efforts to characterize bugs in traditional software systems [130, 126, 131, 132, 133, 134]. Tan et al. [126] conducted an empirical study on three notable projects, including Linux kernel, Mozilla, and Apache, via analyzing around 2k real-world bugs. They revealed that semantic bugs are the major common bugs in general software systems, and memory bugs decrease as they evolve. The significant difference with our analysis is that they did not introduce bug types; instead, they focused on root cause analysis. Also, their analysis is based on general and bug related bugs, while we do not cover the general bugs. Bosu et al. [130] analyzed code review requests from 10 software projects to identify bug code changes. They find that *Race Condition* and *Buffer Overflow* are the most common bug types in traditional software systems. These findings are not aligned with our work where *Race Condition* and *Buffer Overflow* are the least common bug types.

2.5.2 Studies on AI Bugs

Studies on AI API Usage Bugs: Islam et al. [56] conducted the first empirical study on API usage bugs of five DL libraries, including Caffe, Keras, TensorFlow, Theano, and Torch. They collected data from StackOverflow posts, and Github commits to perform their manual analysis. The authors analyzed bug types, root causes, and the impact of bugs in DL libraries and found that data and logic-related bugs are the most common bugs in DL libraries. Zhang et al. [113] studied DL application bugs built on top of TensorFlow and collected bugs from both Stackoverflow and Github projects. They find that fixing patterns and root causes correlate and suggest developers and researchers make automated bug detection approaches on top of root causes. Humbatova et al. [112] provided an extensive and comprehensive taxonomy of faults in DL libraries. They focused on TensorFlow, Keras, and PyTorch for their study. The notable difference between their work with existing studies is that they interviewed 20 researchers and practitioners to increase the reliability of their findings. There are a couple of differences between our work. First, our study merely focuses on Github commits while their study also mined data from Stack overflow posts. Second, they analyzed general bugs of DL libraries while we studied bugs reported in CWE and CVE portals.

Studies on AI Implementation Bugs: The study conducted by Xiao et al. [135] covers bugs in three popular DL libraries including TensorFlow, PyTorch, and Caffe. There are two major differences compared to our work. First, they examine the attack surfaces of DL libraries while we analyze the root cause of bugs in the back-end implementation of AI frameworks. Second, we present a comprehensive study of bugs and identify 16 distinct types of bugs. While they only focus on four potential threats to DL libraries.

Thung et al. [110] studied three popular Java-based AI frameworks to characterize bugs related to the implementation of such tools. They investigated bug reports and bug repositories of the subject programs extracted data from the JIRA issue tracking system, analyzed 500 bugs, and addressed the research questions. There are major differences compared to our empirical study. First, they focus on analyzing bugs in three Java-based AI frameworks while we focus on frameworks that are developed in Python and C/C++. Second, our study focuses on bug types, root causes, symptoms, and fixing patterns while they only focus on the frequency and severity of bugs as well as bug localization. Jia et al. [111] conducted an empirical study on implementation bugs of TensorFlow. More specifically, they targeted more than 36k Github projects that use TensorFlow and extract pull requests, bug reports, and code changes from the corresponding repositories to address the research questions. There are major differences compared to our work. First, they merely focus on the TensorFlow library

while we focus on six more AI frameworks other than TensorFlow that cover state-of-the-art DL models, classic models, and data analysis. Second, we provide 16 different bug types based on CWE number while they merely focus on root causes, symptoms, and fixing patterns. The most related papers to our study are the studies conducted by Franco et al. [116] and Shen et al. [114]. Franco et al. [116] conducted the first study on characteristics of real-world numerical bugs of different numerical libraries, including NumPy, SciPy, LAPACK, GNU Scientific Library, and Elementa. They find that 32% of bugs in the studied libraries are related to *Numeric*. Our study complements their analysis in the sense that ours is more general since we study both numerical and AI frameworks. Shen et al. [114] proposed an empirical study on DL compiler bugs by manually analyzing 595 bugs from Apache TVM, Facebook Glow, and Intel nGraph. The difference between this work and our study is that they focus on DL compiler bugs and provide guidelines for detecting and debugging them, while our study focuses on bugs in AI frameworks and characterizes their types, root causes, symptoms, and fixing patterns.

2.6 Conclusion

This chapter conducts the first empirical study to understand the characteristics of bugs in AI frameworks. The primary motivation behind this study was twofold; characterize AI bugs, and help AI developers design and develop bug detection and debugging techniques to increase the quality and reliability of AI frameworks. We manually analyzed 683 commits from seven widely used AI frameworks to achieve this goal, including TensorFlow, PyTorch, Scikit-learn, Mlpack, Scipy, Pandas, and Numpy. The outcome of this study is 16 bug types, 20 root causes, 4 symptoms, 19 fixing patterns, and six findings. Based on these findings, we further provide a set of actionable guidelines to the AI community to design and develop software bug detection and debugging techniques to increase the reliability of AI frameworks.

Chapter 3

Automatic Static Bug Detection in AI Frameworks

3.1 Introduction

Programming inevitably involves dealing with bugs in software, which is an aspect that can be frustrating for many developers since detecting and fixing bugs is time-consuming [2, 108]. To help developers find software bugs, many static analysis tools have been developed and are now frequently employed by many industries and open-source projects [22, 18, 136, 137]. Error Prone from Google [138], Infer from Facebook [139], and SpotBugs [140], the successor to the widely used FindBugs tool [141], are examples of popular static analysis tools. These tools are usually developed as an analytical framework based on static analysis and are capable of scaling to large applications.

Previous research has examined static analysis tools on traditional projects from different aspects [30, 31, 32, 33, 34, 35]. The major limitation of previous studies is that the datasets used for the empirical evaluation of static analysis tools are not real-world examples, they are not able to replicate new and sophisticated bug patterns. Recently, Lipp et al. [35] addressed the limitations by proposing an empirical evaluation of static analysis tools on real-world datasets collected from CVE records gathered from 27 projects containing 1.15 million lines of code. Their results showed that state-of-the-art tools can detect in-between 20% and 53% of the bugs in a benchmark set of real-world programs. However, it has not been determined if the findings of these studies on conventional projects apply to AI frameworks. Finding real-world bugs in AI frameworks is critical for a couple of reasons. First, AI frameworks have been widely used in many fields in the past decades, such as image

classification [81, 100], big data analysis [101], pattern recognition [102], autonomous driving [103, 104, 105], and natural language processing [88, 106, 107]. Failure to detect bugs in these AI frameworks might have devastating implications, such as traffic accidents [59]. Second, gaining an understanding of the benefits and drawbacks of the static analysis tools that are currently in use will direct future work toward the design of new methodologies or tools for detecting AI-specific bugs.

In this chapter, we set out to investigate the effectiveness of five popular static analysis tools on AI frameworks. Specifically, we conduct an empirical analysis utilizing 410 real-world bugs collected from four widely-used popular AI frameworks to address the issue of how many bugs from these AI frameworks can be detected by static analysis tools and why they miss detecting real-world bugs of AI frameworks. The granularity of our collected dataset is at the commit level in which each commit may contain multiple files involved in fixing the bug. We select five popular and open-source static analysis tools as the research subjects, i.e., Flawfinder [142, 143, 35, 144, 145, 146, 147, 22, 148], RATS [149, 147, 22], Cppcheck [150, 143, 22, 148], Facebook Infer [139, 151, 152, 152], and Clang static analyzer [153, 22, 154, 155, 156]. We run these tools on the modified files from the collected commits in our dataset. For each of these modified files, we extract two versions, i.e., the source code before the bug was introduced and the source code after the bug was fixed. Next, we executed the static analysis tools on the source code before the bug was introduced to identify any warnings or issues present in that version. We follow the diff-based mapping technique used in previous research [157] to identify the warnings that fall within the code changes made during the bug-fixing commit. After that, we proceed to perform a manual inspection of the identified bug candidates. The purpose of this manual inspection is to avoid any potential coincidental matches or false positives. By carefully reviewing the bug candidates, we ensure the accuracy and reliability of our bug detection process.

In addition, we have investigated static analysis tools, using a set of program analysis criteria, such as input representation, pattern matching techniques, and source code element sensitivity. Specifically, we manually examined the documentation and source code of these static analysis tools to understand how they function. We also study the characteristics of bugs in AI frameworks and understand their patterns. As a result, we provide future directions to improve existing static checkers to deal with specific bugs in AI frameworks.

this chapter has the following contributions:

- We present the first empirical study on the investigation of current static analysis tools on four widely used AI frameworks.

- We identify gaps between existing static analysis tools techniques and practices on AI frameworks.
- We explore the capabilities and limitations of each static analysis tool, as well as suggest future ideas for enhancing the effectiveness of these tools in real-world settings to detect bugs specific to AI frameworks.
- We release the dataset of our experiments to help other researchers replicate and extend our study¹⁵.

3.2 Background & Motivation

AI frameworks and traditional software differ in various aspects such as complexity, data dependency, and performance characteristics. Overall, AI frameworks tend to be more complex than traditional software as AI often involves complex mathematical models and algorithms [8, 110, 56, 111]. In addition, as AI models are heavily dependent on the quality and quantity of the data used in their training process, most AI frameworks provide tools for data pre-processing, cleaning, and normalization that are not available in traditional software. AI frameworks are designed to take advantage of modern hardware, including GPUs and specialized processors, to optimize performance. Traditional software, on the other hand, is generally optimized for general-purpose CPUs.

These differences introduce different bug patterns compared to traditional software. Figure 3.1a and Figure 3.1b show two examples of buffer overflow in TensorFlow library¹⁶ and Linux kernel¹⁷ respectively. The root cause of buffer overflow in Figure 3.1a is lack of proper input validation when getting a non-scalar resource tensor while the root cause of buffer overflow in Figure 3.1b is using *snprintf()* instead of using *scnprintf()* as *snprintf()* is not a secure built-in C API. We can see that there are two instances of the same bug (i.e., buffer overflow) that have distinct root causes. To detect a heap buffer overflow example in the TensorFlow library as shown in Figure 3.1a, existing static analysis tools must follow two crucial steps. Firstly, these tools need to recognize the TensorFlow-specific macro checkers (*CHECK_EQ*) by incorporating the macro call signature into their internal database of risky APIs. Secondly, they must conduct data flow analysis to trace the flow of data from client APIs to the backend implementation. In this example, the vulnerable parameter is

¹⁵<https://anonymous.4open.science/r/ISSRE2023SATS-E255/README.md>

¹⁶<https://github.com/tensorflow/tensorflow/commit/13ef0af4867477cdda7e0b294e61560c2952df42>

¹⁷<https://github.com/torvalds/linux/commit/5549af7f42916c0d7e78a0e423ac667e27eaac3e>

```

43 gtl::InlinedVector<npv_intp, 4> dims(ndims);
44 if (TF_TensorType(tensor) == TF_RESOURCE) {
45     dims[0] = TF_TensorByteSize(tensor);
46     CHECK_EQ(ndims, 0)}
47     << "Fetching of non-scalar
48         resource tensors is
49         not supported.";
50     dims.push_back(TF_TensorByteSize(tensor));
51     *nelems = dims[0];
52 } else {
53     *nelems = 1;

```

(a) An example of buffer overflow in TensorFlow library.

```

chip = get_chip_info(sdev->pdata);
for (i=0; i<HDA_EXT_ROM_STATUS_SIZE; i++){
value = snd_sof_dsp_read(sdev, HDA_DSP_BAR,
    chip->rom_status_reg + i * 0x4);
    len += snprintf(msg+len, sizeof(msg)-len,
        " 0x%x", value);
    len += scnprintf(msg+len, sizeof(msg)-len,
        " 0x%x", value);
}
dev_printk(level, sdev->dev,
"extended rom status: %s", msg);

```

(b) An example of buffer overflow in Linux kernel.

Figure 3.1: This figure shows two examples of buffer overflow from the TensorFlow library and Linux kernel.

ndims which has been coming from the client API usage. Unfortunately, we discovered that the current tools cannot perform these essential steps, rendering them inadequate for detecting AI-specific bugs.

The different symptoms and root causes of the same type of bugs from AI frameworks and traditional software motivate us to look deeper into understanding the performance of current static analysis tools on AI frameworks.

3.3 Study Design

In this section, we first discuss the criteria for AI framework selection (Section 3.3.1). Then we elaborate on how we curated a dataset of real-world bugs of AI frameworks (Section 3.3.2). Afterward, we introduce the static analysis tools used in this chapter (Section 3.3.3), and finally, we describe the procedure of applying the static analysis tools to our curated real-world data (Section 3.3.4).

3.3.1 AI Framework Selection

Our study is based on four widely used AI frameworks selected based on a set of inclusion criteria: 1) libraries should be open source and available to the public, 2) they should be under active development, 3) libraries should have the implementation of classical frameworks as well as state-of-the-art DL models, 4) libraries should support different tasks in the common AI workflow. The outcome of applying these criteria is the following AI frameworks; TensorFlow [71], PyTorch [72], MXNet [73], and Mlpack [74]. Our filters also exclude some well-known libraries including Caffe, Theano, and Keras because the data extraction mechanism utilized in this work did not retrieve enough real-world bugs from their repositories due to a lack of sufficient bug records in the GitHub repository of these libraries.

3.3.2 Collection of Bugs from AI Frameworks

For each of the studied AI frameworks, we followed the approach proposed by Zhou et al [122] to extract bug-related commits. Note that we extract all commits in the default branch of each library since the starting date of the development. Specifically, we use their regular expression rules, including expressions and keywords related to bugs, to collect bug-fixing commits. As a result, we collected around 5k commits. Note that, the collected commits might contain noises because there may be coincidental matches between bug keywords and the keywords inside the commit message and title [122]. To remove noises, we further conducted a manual inspection on each commit. In our manual inspection, two authors began evaluating extracted commits simultaneously. The authors analyzed the title, message, merged pull requests, and linked issues¹⁸ of each bug fixing commit and remove the commits that are not related

¹⁸Some bug fixing commits fix opened issues. Thus, the authors further analyzed them for noise removal.

Table 3.1: Characteristics of bug fixing commits (BFCs) used in this chapter.

Library	Language	LOC($\sim$)	# VFCs
Mlpack	C++	340K	47
MXNet	C++/Python	362K	60
PyTorch	C++/Python	3.8M	58
TensorFlow	C++/Python	567K	245
Overall		5M	410

to a registered bug in the CWE website¹⁹. As a result, we collected 410 bug-fixing commits across the four projects that were investigated (shown in Table 3.1).

In this chapter, we use the term *bug* in a general sense to refer to any kind of software defect, including bugs, logical bugs, and performance bugs.

3.3.3 Running Static Analysis Tools

In this chapter, we select five widely used and open-source static analysis tools including Facebook Infer [139], Clang static analyzer [153], Cppcheck²⁰, Flawfinder (Flawfinder [158], and RATS²¹). The selection of five tools used in our study is based on a comprehensive review of recently published papers on the empirical evaluation of static analysis tools [33, 35]. Please note that three of the tools are under active development including Facebook Infer [139], Clang static analyzer [153], and Cppcheck²². The two remaining tools (Flawfinder [158] and RATS²³) are not actively developing but they have been frequently used in bug detection [22, 159, 160].

The static analysis tools used in this work have several common characteristics. For example, Flawfinder and RATS treat the input source code as a text sequence while Cppcheck, Infer, and Clang static analyzers convert the source code into an intermediate representation. All tools use a built-in database of patterns that are used to detect bugs. For instance, RATS, Flawfinder, and Cppcheck all use a database of C/C++ system functions that are known to have bugs such as buffer overflow or format string issues. while Infer uses a bi-abduction inference and analysis to find bug source code statements. Clang static analyzer works by parsing the source code and checking it against a set of predefined checks, or *linters*.

¹⁹<https://cwe.mitre.org/>

²⁰<https://cppcheck.sourceforge.io/>

²¹<https://github.com/andrew-d/rough-auditing-tool-for-security>

²²<https://cppcheck.sourceforge.io/>

²³<https://github.com/andrew-d/rough-auditing-tool-for-security>

Table 3.2: Characteristics of tools based on different program analysis factors.

Tool	Version	Input representation			Pattern matching		Sensitivity				
		Text	AST	Other	Intraprocedural	Interprocedural	Context	Field	Object	Data Flow	Control Flow
Flawfinder	2.0.19	✓	-	-	×	×	×	×	×	×	×
RATS	2.4	✓	-	-	×	×	×	×	×	×	×
Cppcheck	2.7	-	✓	-	✓	×	✓	×	×	*	*
Infer	1.1.0	-	-	✓	✓	✓	✓	✓	N.A	✓	✓
Clang static analyzer	14.0.0	-	✓	-	✓	✓	✓	✓	✓	✓	✓

For the capabilities of these five static analysis tools used in this chapter, following existing work [34], we take into consideration the following three different types of program analysis features, i.e., input representation, matching strategies (*intraprocedural* and *interprocedural*), and sensitivity to program elements including *flow*, *context*, *field*, *object*, *path*, and *field*. We investigated the tools concerning the aforementioned analytical program analysis features. During this procedure, we manually evaluated the tools’ source code and documentation, which demonstrated several types of behaviors to check the tools’ capabilities and limits. In the following paragraphs, each tool is explained in detail. Table 3.2 shows the capabilities of static analysis tools investigated.

Flawfinder [142]: Is a static analysis tool that scans a program for potential bugs by using a database of known unsafe C/C++ functions. Flawfinder can detect problems with race conditions and system calls in addition to `printf()` and normal string manipulation operations, based on an internal database that contains C/C++ routines that are known to have bugs in their design.

RATS [149]: An open-source static analyzer that can analyze code bases written in C, C++, Perl, PHP, and Python. Similar to Flawfinder, it uses an internal database of risky C/C++ API signatures and uses a keyword-matching approach to find and mark them as bugs in the target source code.

Cppcheck [150]: Cppcheck offers one-of-a-kind code analysis to find defects and focuses on finding undefined behavior as well as risky coding structures. The objective is to generate an extremely low number of false positives.

Infer [139]: Is a static analysis tool that is developed by Facebook. It searches for a wide variety of bugs in programs written in Java, C/C++, and Objective-C. Bi-abduction analysis is one of the methods that Infer employs to locate bugs such as deadlocks, memory leaks, and null pointer dereference. To perform analysis, Infer requires a set of compilation commands for each file. Infer is an interprocedural analysis tool which means that it allows keeping track of objects and variables between methods, and also global variables.

Clang Static Analyzer [153]: Clang static analyzer is built on top of the LLVM project, which offers a set of modular and reusable compiler and toolchain technologies. It is an extensible framework for C/C++ code linting that may be used to enforce coding standards, conduct static analysis, and discover probable errors. It operates by scanning the source code and comparing it to a collection of predefined checks, known as linters. The checks are built as separate modules, making it simple to add new checks or modify current ones.

Note that the static analysis tools used in this chapter have capabilities to report non-bug related warnings, e.g., styling or refactoring issues, as well as reporting general bugs that are not related to bugs that have a unique id in CWE website²⁴. To reduce false positive rates and prevent static checkers from producing related warnings to bugs, we have configured them to merely report warnings that are strongly related to bugs. Configuring the tools to report all types of bugs can introduce a large number of false positives (non-bug-related code), which require much more time for manual verification. Please note that RATS does not generate styling issues. In addition, we manually studied Flawfinder’s built-in database and discovered that all rules have a unique ID associated with CWE records. As a result, we solely used this setup for Infer, Cppcheck, and Clang static analyzer.

3.3.4 Identification of Bug Candidates

One of the main challenges in the identification of bug candidates is to check whether the reported warnings by static analysis tools correspond with the issued bug class or not. To identify bug candidates, we use the following steps. First, we performed automatic filtering based on the commits’ diff information as used in existing studies [33, 34]. Afterward, we further analyzed the reported warnings that involve the commit using manual inspection.

Automatic Filtering: We filter reported warnings by using a diff-based mapping technique which uses code diffs between bug and fixed programs in a fixing commit [157]. First, it computes a set of lines in the bug program that is flagged with at least one warning. Then, it checks whether the flagged line numbers overlap with the changed lines in the fixing commit. If the flagged lines overlap with the code change, the warning is considered a bug candidate. Otherwise, it is not a candidate for manual inspection. For example, Figure 3.2 shows an example of a warning generated by Flawfinder that is considered to be a bug candidate since the line number reported in the warning overlaps with the line number in the code change of the bug fixing commit.

²⁴<https://cwe.mitre.org/>

```

43  for (int i = 0; i < client->device_count(); ++i) {
44  se::StreamExecutorConfig config;
45  config.ordinal = i; config.device_options.non_portable_tags
46  ["host_thread_stack_size_in_bytes"] =
47      |{\color{RGB}{255, 99, 71}-absl::StrCat(2048 * 1024);}|
48      |{\color{RGB}{0, 153, 76}+absl::StrCat(8192 * 1024);}|
49  TF_ASSIGN_OR_RETURN(se::StreamExecutor * executor,
50  platform->GetExecutor(config));

```

(a) Bug fixing commit.

```

Examining /cpu_device.cc
FINAL RESULTS:
cpu_device.cc:47:
[4] (buffer) StrCat:
Does not check for buffer overflows when
concatenating to destination
[MS-banned] (CWE-120).
absl::StrCat(2048 * 1024)

```

(b) Reported warning by Flawfinder.

Figure 3.2: An example of bug fixing commit in TensorFlow library in which the developer has increased the thread stack size from 2048 to 8192 to prevent buffer overflow bug. Figure 3.2b shows the generated warning by Flawfinder in which there is a possible buffer overflow at line 47.

Manual Verification: In this step, we manually scan all candidates and check the warning messages against the bug and clean versions of the code to eliminate potential accidental matches. The first two authors examine the warnings and bug-fixing commits simultaneously. They manually analyze the warning message, the CWE-ID, and the line number where the bug is reported. Based on the above information, they review the code changes to confirm whether the warning is related to the bug being addressed in the commit. Note that, in the cases where the static analysis tools reported a warning at a line outside those changed lines in the bug-fixing commit, we manually checked the warning. In addition, we have also performed a backward analysis in which we traced all variables and function calls within the code changes up to the marked line that was outside the code changes. If the marked line was associated with the code changes, we considered the corresponding warning

as a potential bug. If there is any disagreement, they flag the commit and the corresponding warning for further manual verification in the next round. They repeat this process several times until all warnings and commits have been reviewed.

During our manual inspection, we also audit the following information: 1) True Positive (TP): The bug described in the warning precisely matches with the bug reported for the commit; False Negative (FN): A potential warning mistakenly predicted to be a false alarm. In this chapter, we calculate the True Positive Rate (TPR) and False Negative Rate (FNR) as $TPR = \frac{TP}{TP+FN}$ and $FNR = \frac{FN}{FN+TP}$ respectively.

3.4 Experimental Results

In this section, we present and discuss our analysis results to address the following three research questions.

RQ1 (Detection Capability): How many warnings are reported by the studied static analysis tools?

RQ2 (Detection Effectiveness): How effective are static analysis tools at detecting real-world bugs in AI frameworks?

RQ3 (Root Cause): What are the root causes of missing real-world bugs in AI frameworks?

To answer these RQs, we execute Flawfinder, RATS, Cppcheck, Infer, and Clang static analyzer on our dataset of 410 real-world bugs across the four studied AI frameworks. The output of bug detectors, i.e., generated warnings, are automatically parsed to extract relevant information including bug types, line numbers, and warning messages. All data and scripts for conducting the experiments described in this section are accessible to the public.

3.4.1 RQ1: Detection Capability

Experiment Setup: To answer this question, we directly run these five static analysis tools on the four AI frameworks, we record all the reported bugs (including the detailed location and CWE information) by these tools for further analysis. For each of the reported bugs, we further manually check whether it’s a true bug or a false positive. Please note that the granularity of our collected dataset is at the commit level in which each commit may contain multiple files involved in fixing a bug. For each commit, we extract modified files, i.e., affected by the bug. For each modified file, we extracted two versions, i.e., source code before the bug and source code after

Table 3.3: Distribution of reported warnings.

Library	Flawfinder	RATS	Cppcheck	Infer	Clang static analyzer
Mlpack	3	22	1	0	0
MXNet	10	6	7	0	0
PyTorch	5	1	2	4	16
TensorFlow	50	139	0	2	254
Overall	68	168	10	6	270

fixing the bug. We then run the tools on the source code before the bug and generate the warnings.

Table 3.3 displays the number of bug types extracted from warnings reported by each static analysis tool across the four AI frameworks. Specifically, among the four AI frameworks, TensorFlow has the highest number of reported warnings, i.e., in total 445 warnings. *Mlpack* has the lowest number of reported warnings overall, with only 26 warnings across all tools. Among the static analysis tools, *Clang static analyzer* reports the highest number of warnings for each library. *Infer* only reports six warnings.

We further illustrate the top 10 bug types and their distribution in our curated dataset in Figure 3.3. The chart shows that the most common bug type is CWE-190 (Integer Overflow or Wraparound), with 86 occurrences in the studied AI frameworks. The next most common types are CWE-362 (Race Condition), CWE-401 (Missing Release of Resource after Effective Lifetime), and CWE-476 (NULL Pointer Dereference), with 45, 43, and 34 occurrences, respectively. The least common bug type in the chart is CWE-122 (Heap-based Buffer Overflow), with only 16 occurrences.

Table 3.4 also shows the number of false positive warnings out of all warnings generated given each true bug type. Specifically, Flawfinder reported a total of 50 warnings for CWE-120, but only 3 of them were accurate warnings. In addition, it reported a single warning for CWE-190 that was indeed accurate. For CWE-20, Flawfinder reported 5 true positive warnings. Regarding CWE-190, Flawfinder reports one truly positive warning. RATS produced 27 warnings for CWE-121, but only 1 of them is a true warning. For CWE-122, RATS flagged 8 warnings, but merely 3 of them are true positive. All warnings generated by Cppcheck and Infer are false positives suggesting that they are not effective at detecting real-world security bugs in the studied AI frameworks.

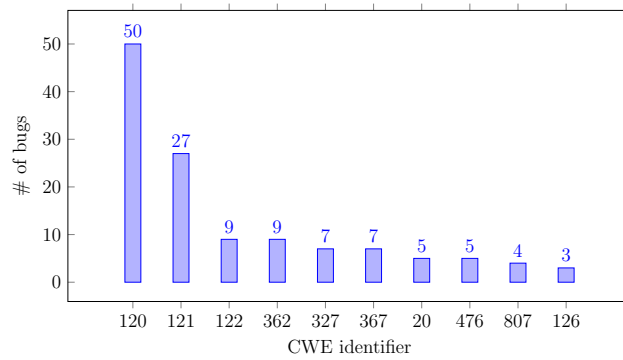


Figure 3.3: The top 10 bug types in our dataset.

Finding 1: The number of warnings produced by the five static analysis tools varies significantly and is mainly determined by their detection mechanisms. While these tools can also generate false positives, human review is still necessary to determine whether reported warnings are real-world bugs that require attention.

Table 3.4: Distribution of bug types reported by the four static bug detectors. The warnings generated by the Clang static analyzer do not have a CWE number, so we exclude it in this table. Numbers in the brackets are false positive warnings. The numbers outside of the brackets are the total number of generated warnings.

CWE-ID	# True Bugs	Flawfinder	RATS	Cppcheck	Infer	Overall
CWE-120	-	50 (47)	0	0	0	50
CWE-121	25	0	27 (26)	0	0	27
CWE-122	24	0	8 (5)	0	1 (1)	9
CWE-362	45	7 (7)	2 (2)	0	0	9
CWE-327	-	7 (7)	0	0	0	7
CWE-367	-	7 (7)	0	0	0	7
CWE-20	7	5 (0)	0	0	0	5
CWE-476	34	0	0	5 (5)	0	5
CWE-807	-	4 (4)	0	0	0	4
CWE-126	-	3 (3)	0	0	0	3
CWE-330	-	0	3 (3)	0	0	3
CWE-457	1	0	0	0	2 (2)	2
CWE-563	-	0	0	0	1 (1)	1
CWE-667	-	0	0	1 (1)	0	1
CWE-758	-	0	0	1 (1)	0	1
CWE-190	86	1 (0)	0	0	0	1
CWE-398	-	0	0	1 (1)	0	1
CWE-119	-	1 (1)	0	0	0	1
CWE-833	15	0	0	0	1 (1)	1
CWE-401	43	0	0	0	0	0
CWE-908	27	0	0	0	0	0
CWE-835	26	0	0	0	0	0
CWE-1331	18	0	0	0	0	0
CWE-125	17	0	0	0	0	0
CWE-369	10	0	0	0	0	0
CWE-703	4	0	0	0	0	0
CWE-191	3	0	0	0	0	0
CWE-705	3	0	0	0	0	0
CWE-415	3	0	0	0	0	0
CWE-704	2	0	0	0	0	0
CWE-416	2	0	0	0	0	0
CWE-840	2	0	0	0	0	0
CWE-787	2	0	0	0	0	0
CWE-439	2	0	0	0	0	0
CWE-628	1	0	0	0	0	0
CWE-241	1	0	0	0	0	0
CWE-255	1	0	0	0	0	0
CWE-197	1	0	0	0	0	0
CWE-252	1	0	0	0	0	0
CWE-706	1	0	0	0	0	0
CWE-1006	1	0	0	0	0	0
CWE-561	1	0	0	0	0	0
CWE-475	1	0	0	0	0	0
Overall	410	85	40	8	5	138

3.4.2 RQ2: Detection Effectiveness

Experiment Setup: To address this question, the first two authors of the study manually reviewed the generated warnings per file for each bug-fixing commit. The team conducted a thorough analysis of the warning message, the CWE-ID, and the specific line numbers that can indicate where the bug was identified. Using this information, they further examined the code modifications to verify that the warning was indeed related to the bug being addressed in the particular update. If any discrepancies arise, they highlight the corresponding warning and commit to additional examination in the subsequent review cycle. This process is repeated until all the warnings and code changes have been assessed.

Table 3.5: Performance of the subject static analysis tools across four AI frameworks.

Tool	TPR	FNR
Flawfinder	0.04	0.95
RATS	0.03	0.97
Cppcheck	0	1
Infer	0	1
Clang static analyzer	0	1

Table 3.6: Bugs detected by Flawfinder and RATS.

Tool	Actual bug	Reported bug(# warnings)	Library	Filename
Flawfinder	CWE-787	CWE-20(4)	PyTorch	<i>simd.h</i>
Flawfinder	CWE-197	CWE-120(2)	PyTorch	<i>decode_padded_raw_op.cc</i>
Flawfinder	CWE-122	CWE-20(1)	MXNet	<i>image-classification-predict.cc</i>
Flawfinder	CWE-190	CWE-190(1)	MXNet	<i>image_iter_common.h</i>
Flawfinder	CWE-121	CWE-120(1)	TensorFlow	<i>cpu_device.cc</i>
RATS	CWE-122	CWE-121(1)	MXNet	<i>image-classification-predict.cc</i>
RATS	CWE-190	CWE-122(1)	PyTorch	<i>THTensor.cpp</i>
RATS	CWE-197	CWE-122(2)	PyTorch	<i>decode_padded_raw_op.cc</i>

Results: Table 3.5 shows the performance of each static analysis tool regarding True Positive Rate (TPR) and False Negative Rate (FNR) values. As we can see from the table, Flawfinder has a TPR of 0.049, which means it correctly identifies only 4.95% of the actual bugs, and an FNR of 0.95, indicating that it fails to detect 95.05% of actual bugs. RATS has a TPR of 0.030 meaning that it correctly identifies only 3.07% of the actual bugs, and a FNR of 0.97, which means it fails to detect 97.01% of the actual bugs. The rest of the tools have a TPR of 0 which states that they fail to detect any actual bugs, and an FNR of 1, which means they incorrectly identify all actual bugs as negatives.

Overall, all the experimented tools have relatively low performance in terms of identifying actual bugs in AI frameworks, as they all have very low TPR values or fail to detect any real-world bugs at all.

Table 3.6 shows the detailed characteristics of bugs detected by Flawfinder and RATS. We can see that Flawfinder and RATS can identify bugs in different code repositories, including MXNet, TensorFlow, and PyTorch. The bugs are categorized according to their types (i.e., CWE numbers). For example, Flawfinder detected a CWE-122 and a CWE-190 in MXNet. RATS also detected a CWE-122 in MXNet, as well as a CWE-190 in PyTorch.

```

84     auto input_shape = c->input(0);
85     auto input_h_shape = c->input(1);
86     auto seq_length = c->Dim(input_shape, 0);
87     // assumes rank >= 2
88     auto batch_size = c->Dim(input_shape, 1);
89     // assumes rank >= 3
90     auto num_units = c->Dim(input_h_shape, 2);

```

Figure 3.4: An example of heap buffer overflow in TensorFlow library.

Finding 2: Overall, the effectiveness of the tools is quite poor in discovering real-world ML software bugs. Flawfinder and RATS, which are the most effective static checkers, discovered 4 unique bugs out of a total of 410 bugs in our dataset.

3.4.3 RQ3: Root Cause of Missing Real-world Bugs

To address this research question, we manually analyzed and reviewed the documentation of tools as well as their code base. Then, we organized the reasons why these tools miss detect so many bugs.

Flawfinder and RATS: Since Flawfinder and RATS manifest very similarly in bug detection, Flawfinder’s specific reasons also apply to RATS. **Issues with Soundness Strategy:** Flawfinder and RATS have a significant limitation in their approach to bug detection, known as the soundness strategy. The soundness strategy of Flawfinder and RATS is that they use pre-defined risky C/C++ APIs in their internal databases to search the source code and label every matched API call (already registered in the database) as a potential bug. Although AI frameworks often make extensive use of C/C++ APIs in their backend implementation, the bugs in these libraries do not primarily originate from these APIs. Instead, the bugs in AI frameworks tend to exhibit more complex patterns (require a deeper source code analysis to be detected), which are not effectively captured by the simplistic approach of flagging all API calls as vulnerable [8, 56, 110, 135, 110, 111, 116, 114].

Lack of ML-specific Vulnerable API Information: Even though Flawfinder and RATS support more than 200 dangerous C/C++ APIs, they are still incapable of supporting ML-specific vulnerable APIs. One of the reasons they miss detecting real-world security bugs in AI frameworks is that they do not model library-specific API information [8, 111]. For example, to detect *Memory Leak (CWE-401)* in the

Mlpack library, the static detectors need to model `CleanMemory()` or `delete_mat` which are Mlpack-specific APIs used for cleaning allocated memories.

Lack of Data Flow and Control Flow Support: Flawfinder and RATS face challenges in detecting numerous bug patterns specific to machine learning (ML) because they cannot model control flow and data flow. Instead, these tools rely on a simplistic keyword-matching approach to identify potentially dangerous C/C++ API calls and flag them as bugs. This limitation hinders their effectiveness in capturing the intricacies of ML-specific bugs [111, 8] that involve complex control and data dependencies. For example, lack of validation is a complex root cause pattern that is the major root cause of data type bugs in AI frameworks [8]. In this particular bug pattern, bugs can arise when utilizing client APIs of AI frameworks. If the backend implementation fails to validate or properly handle malicious inputs or malformed values received through these APIs, bugs can occur. It is crucial to perform appropriate validation and handling of such inputs to mitigate potential issues and ensure the robustness and security of the ML library. For example, Figure 3.4 is an example of a lack of validation that causes a heap buffer overflow. In this bug, the code assumes `input_shape` and `input_h_shape` have a specific rank, while the rank values should be validated to avoid possible invalid memory access. Flawfinder and RATS are not able to detect such bugs because they are incapable of modeling data flow dependency and keeping track of `input_shape` and `input_h_shape` to check if these variables are validated or not.

Cppcheck Limited Buffer or Stack Overflow Checking: Existing work [8] has shown that the root cause patterns of buffer overflow or stack overflow in AI frameworks are significantly different compared to that of traditional software. For example, in this commit²⁵ from the TensorFlow library, the root cause of stack overflow is a very big computation graph of functions and edges. In this example, Each instance of the class `std::shared_ptr<Function>` holds a collection of Edge objects, and each Edge object, in turn, holds a `std::shared_ptr<Function>`. Removing a `std::shared_ptr<Function>` can lead to the cascading deletion of other `std::shared_ptr<Function>` instances, potentially resulting in a stack overflow if the graph has a significant depth. However, Cppcheck performs different strategies to detect buffer overflow and stack overflow bugs which are not strong enough to detect ML-related overflow bugs. For example, Cppcheck uses array index checkers to detect buffer overflow which identifies array index operations and performs various checks to detect potential buffer overruns associated with array indexing. More specifically, Cppcheck exhaustively searches for array indexing statements in the code snippet, even if the code is not reachable, without any control flow analysis. This technique is

²⁵<https://github.com/tensorflow/tensorflow/commit/932c4c2364884af52609ea8a86c7232a926d958f>

incapable of detecting overflow bugs which is due to the large computation graph mentioned above. The second technique to detect buffer overflow or stack overflow is to search for C/C++ API calls, similar to Flawfinder and RATS explained earlier. In this technique, Cppcheck examines the API scopes and the corresponding arguments, analyzes the buffer or stack sizes associated with the arguments, and detects potential buffer overflow or stack overflow bugs based on specified minimum size requirements.

Limited Control Flow Analysis: One major limitation of Cppcheck is limited control flow analysis, while a strong control flow analysis is required to detect very hard-to-detect bugs. In AI frameworks, e.g., memory leaks that have complicated bug patterns. For example, this memory leak bug²⁶ occurs in the TensorFlow library when decoding malformed PNG image. The memory leak occurs when certain errors in the function implementation cause the execution to be abruptly terminated using the `OP_REQUIRES` macro checker. This termination prevents the proper freeing of allocated buffers stored in the `decode` value. To release these allocated buffers, the function should call `png::CommonFreeDecode(&decode)`. However, due to eager termination, the necessary memory-freeing process is not allowed to occur. This bug is very hard to detect by Cppcheck for two main reasons. First, the TensorFlow-specific API `png::CommonFreeDecode(&decode)`; is not registered in the internal database of API symbols which is the first step toward detecting this leak. Second, the control-flow analysis supported by Cppcheck is very simple and cannot capture the complex flow analysis in this bug pattern.

Infer Limited Buffer Overflow Checkers: Compared to Cppcheck, Flawfinder, and RATS, Infer has a stronger buffer overflow checker. Infer uses symbolic intervals to handle the range of index values and buffer sizes. Typically, interval analysis involves working with intervals represented as `[low, high]`, where `low` and `high` are constants indicating the lower bound and upper bound of the target buffer and indexing range. However, this checker is not effective enough to detect ML-related buffer overflow bugs. For example, Figure 3.5 shows an example of hard-to-detect buffer overflow²⁷ in the TensorFlow library where providing a tensor with larger dimensions as the second argument is critical. To address this issue, the developer has modified the `MatchingDim` function to return the minimum size between the two dimensions.

Clang static analyzer Limited Rules: Although the Clang static analyzer provides a large number of predefined rules (i.e., there are 25 families of rules implemented in the Clang static analyzer database), it does not cover rules of security bugs that are relevant to AI frameworks. For example, the null pointer checker introduced

²⁶ <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23585>

²⁷ <https://github.com/tensorflow/tensorflow/commit/8ee24e7949a203d234489f9da2c5bf45a7d5157d>

```

84 inline int MatchingDim(const RuntimeShape&
85 shape1, int index1, const RuntimeShape&
86 shape2, int index2) {
87     TFLITE_DCHECK_EQ(shape1.Dims(index1),
88     shape2.Dims(index2));
89     return shape1.Dims(index1);
90 }

```

Figure 3.5: An example of hard-to-detect buffer overflow bug in TensorFlow library.

in Clang static analyzer, i.e., *core.NullDereference* (*C*, *C++*, *ObjC*) is only able to detect simple null pointer violations²⁸. While, in a null pointer bug from TensorFlow library²⁹, the root cause is lack of checking null pointer Tensors as a function argument. Specifically, the bug traces back to the omission of null pointer checks about Tensors utilized as function arguments. Within this instance, the implicated function lacks a TensorFlow-specific mechanism to preempt the erroneous passage of *val* to the (*GetBundleEntryProto(key, &entry)*). In the fixing commit, the developers add *CHECK(val != nullptr)*; to resolve the issue.

An additional instance of a constrained rule within the Clang static analyzer is the *unix.Malloc* (*C*) checker. This checker’s role is to identify memory leaks within codebases developed in the C programming language. The mechanics of this Clang static analyzer checker, elaborated upon in the official documentation³⁰, allow it to uncover memory leaks by employing relatively straightforward patterns. Nevertheless, an illustration of a memory leak within the MXNet library³¹ serves to underscore the complexity that can underlie memory leak patterns.

```

std::unique_ptr<DType>
tmp(new DType[Super::WORKLOAD_COUNT]);
std::unique_ptr<DType[]>
tmp(new DType[Super::WORKLOAD_COUNT]);

```

²⁸<https://clang.llvm.org/docs/analyzer/checkers.html#core-nulldereference-c-c-objc>

²⁹<https://github.com/tensorflow/tensorflow/commit/ca170f34d9174d6981850855190a398393aa921e>

³⁰<https://clang.llvm.org/docs/analyzer/checkers.html#unix-malloc-c>

³¹<https://github.com/apache/mxnet/commit/b1ad1619d8007dc7dfc0f4ba9cc8720b3e0dd32a>

Finding 3: We identified a set of specific reasons that help explain why the five static analysis tools examined in this chapter miss real-world security bugs in AI frameworks including ML software-specific reasons, i.e., *Lack of Implementation Support of Detection Rules*, *Issues with Soundness Strategy*, and *Lack of ML-specific Vulnerable API Information*.

3.5 Lessons Learned

Our study reveals several interesting findings that can serve as applicable guidelines for improving static analysis tools for AI frameworks.

3.5.1 Implication for Improving Flawfinder and RATS

We find that Flawfinder and RATS flag every C/C++ API as a bug regardless of whether they are bugs or not. This behavior introduces many false alarms 3.4. To reduce the false alarm rate, numerous extensions to these checkers are required.

Extend the Input Representation: Flawfinder and RATS represent the source code as a sequence of tokens. Unlike natural language, source code encodes structural information [161], which needs to be considered with rich representation techniques like Abstract Syntax Trees(AST), Control Flow Graphs(CFG), or Data Flow Graphs(DFG) [162]. Having a control flow graph is vital in detecting many AI-related bugs including buffer overflow and memory leaks. Control flow graphs allow the tool to model every possible execution patch inside buggy programs. For example, to detect the stack overflow bug in this commit ³², the checker needs to traverse control graph to find out that the implementation of `DataTypeString` function is buggy due to missing `return DataTypeStringInternal(dtype);`.

Extend the Pattern Matching: The pattern-matching strategy of Flawfinder and RATS follows naive text-based matching. Text-based matching does not allow the detectors to perform intraprocedural analysis, i.e. the analysis inside the boundary of functions or any compilation units. The lack of intraprocedural analysis introduces so many false alarms in the case of detecting AI bugs. For example, to detect *Memory Leaks (CWE-401)* bug listed in Table 3.6, the detector needs to perform analysis inside `_bsplmat()` and catch the return value which is the root cause of this bug.

³²<https://github.com/tensorflow/tensorflow/commit/698bc996f7190f5cd836d48d29b8c1b3ddcd37c2>

3.5.2 Implication for Improving Cppcheck

Cppcheck is the most effective static checker for detecting real-world bugs. However, it still lacks a plethora of weaknesses. Cppcheck may be extended in two ways, as discussed in the subsections below.

Support AI Framework-Specific Constructs: To extend Cppcheck, the developers should handle any AI framework-specific constructs or macros within the backend implementation. This may involve extending Cppcheck’s macro handling capabilities or creating additional checks to handle these constructs effectively.

Extend the Control Flow Graph: The control flow graph analysis in Cppcheck is very limited, with the following assumptions: all source code statements can be accessed, and the state checkers in the if conditions are always either true or false. As a result, a more advanced control flow graph is required to be able to detect sophisticated bugs in AI frameworks.

3.5.3 Implication for Improving Infer

Improve Buffer Overflow Checker: While Infer’s buffer overrun checker can detect typical buffer overflow patterns, it may not cover all conceivable variations and attacker approaches in AI frameworks. The tool may miss sophisticated or unique exploitation methods explained in RQ3.

Isolate Checkers: To begin with, the rules for identifying integer overflow in Infer are still in an experimental stage and lack sensitivity to function arguments. Additionally, Infer’s dependency on buffer overrun rules to detect integer overflow introduces a limitation. These buffer overrun rules themselves tend to generate a significant number of false alarms, which consequently raises the false alarm rate for identifying integer overflow bugs. Given the intricate nature of the bug pattern for integer overflow in AI frameworks [8], it becomes crucial for developers to separate the buffer overrun checkers from the scope of integer overflow.

3.5.4 Implication for Improving Clang static analyzer

Extend the Checker for Null Pointer Dereference: Clang static analyzer uses its internal checker *core.NullDereference* (*C*, *C++*, *ObjC*) to detect null pointer dereference bugs in C, C++, and Objective C programs. The checker works well if function arguments take pointer parameters. If the code attempts to dereference the pointer without checking if it is null, it marks it as buggy. However, in terms of AI

frameworks, null pointer dereference has more sophisticated patterns. For example, in this commit from the TensorFlow library³³, the developer has removed:

```
int64 id = ctx->session_state()->GetNewId();
```

Since `session_state()` is prone to having a null value which results in denial of service via a null pointer dereference. The developers should increase the null pointer checker to cover more corner cases in terms of code elements that may have null values.

3.6 Threats to Validity

As is the case with every empirical research, there are a few factors that call into question the reliability of the inferences we have taken from our data. One of the major limitations is the choice of AI frameworks and static analysis tools. To protect ourselves against this risk, we have chosen four extensively used AI frameworks, each of which focuses on a different facet of AI development. All AI frameworks are open-source projects that are always being developed and improved upon. Regarding the selection of static analysis tools, our major focus is on static detectors that are openly available to the public and are in the process of being developed right now. We used widely used and popular detectors which have been cited by many previous studies in the field of bug detection [22, 159, 160].

Another possible threat to this chapter is the mapping technique [33] used to find potential vulnerable candidates. The approach we used in the chapter is subject to coincidental matches. For example, the assumption is that if the line number produced in warnings overlaps with the modified lines in a bug-fixing commit, the program will automatically recognize the warning as a candidate for the bug. This is not true in practice since the reported warning may not be connected to the actual bug in the commit, or the changed line may be refactoring the code and does not represent the bug being fixed. The ultimate determination of whether a warning relates to a bug is made by two authors involved in manual inspection and is therefore subjective. To mitigate this risk, both authors reviewed every possibility for an identified bug when there is no clear evidence.

In this chapter, we rely on the assumption that a source code before a bug fix is a vulnerable source code, and the source code after the fix is considered as the bug is fixed. One may apply static analysis tools to source code after the fix and find multiple bugs. As a result, this assumption serves as the foundation for this chapter's

³³<https://github.com/tensorflow/tensorflow/commit/9a133d73ae4b4664d22bd1aa6d654fec13c52ee1>

goal of determining how many real-world bugs the tools identify at the moment of committing modifications.

3.7 Related Work

3.7.1 Bug Detection

There have been numerous studies focused on bug detection in the literature [18, 26, 22, 160, 159, 136]. Cao et al. [18] proposed a deep learning-based bug detection model to detect memory-related statements on their manually curated dataset extracted from 11 projects developed in C/C++. Their proposed model can model structural information which allows the detection of semantic bugs. Their experiments indicate that the proposed model is superior compared to cutting-edge models as well as static analysis tools. Li et al. [26] developed SySeVR, a deep learning-based bug detector in which syntactic and semantic information contained in source codes are merged as a rich input representation and supplied into the model. They think that this unique representation discovers subtle weaknesses in the source code.

3.7.2 Studies on Static Detection Techniques

Habib and Pradel [33] investigated the static analyzers including Infer, ErrorProne, and SpotBugs to determine what percentage of Defect4j’s total bugs can be located using the aforementioned tools. The authors utilized both the code diff and the bug report mapping approaches. Tomassi [163] They carry out a study in which they examine the similarities and differences between ErrorProne and SpotBugs to determine the total number of bugs that are discovered in a sample of 320 BugSWARM artifacts. SpotBugs was only able to locate a single bug, as the author discovered. Rutar et al. [164] analyzed a small suite of programs with several different static analyzers, including PMD, FindBugs, JLint, Bandera, and ESC/Java 2. The authors present a taxonomy of bugs discovered by each tool, demonstrating that none of the tools can be considered to be more comprehensive than the others. Runtime as well as the total number of warnings generated are the primary focuses of this study. Lipp et al. [35] argued that there are two main limitations to the datasets used for testing static analysis tools. Firstly, the datasets do not accurately represent real-world code and cannot identify new and complex bug patterns. Secondly, the datasets do not classify bugs based on the Common Weakness Enumeration (CWE) mapping. Hence, they attempted to overcome these limitations by testing static analysis tools

on real-world datasets that were collected from CVE records of 27 different projects, totaling 1.15 million lines of code.

Static analysis tools often produce a large number of false alarm warnings which makes manual inspection problematic. To remove such false alarms, Several different approaches to the detection of bugs have been taken up by the industry. Bessey et al. [165] share their insights gained from the process of bringing static analysis tools to the market in their paper. Ayewah et al. [141, 166] discussed the lessons learned via applying FindBug on Google’s codebase in which numerous engineers involved through thousands of FindBugs generated warnings, and addressed them by either fixing them or filing reports. They found that most issues were highlighted for fixing, but only a few of them were causing significant problems in production. Understanding real-world bugs is a crucial first step in enhancing bug detection. Several studies have taken different types of bugs into account, such as those in the Linux kernel [167], bugs in concurrency [168], and bugs in correctness [169] and performance [170] in JavaScript.

Our work is very different from the studies that have been done before [35, 33, 163, 34]. In previous studies, the primary focus has been on the automatic static detection of general bugs in traditional software projects. On the other hand, the primary focus of this chapter is on the automatic detection of bugs in widely used AI projects. For example, Integer Overflow (CWE-190) is a major bug in AI frameworks as found in [8]. A further distinction lies in the fact that the focus of our work is on the more recent and advanced generation of static analysis tools specifically designed to detect bugs in projects written in C/C++.

3.8 Conclusion

This chapter addresses the critical task of automatic detection of bugs in AI frameworks. The study analyzes the effectiveness of five popular and widely used static analysis tools, namely Flawfinder, RATS, Cppcheck, Facebook Infer, and Clang static analyzer, on a curated dataset of bugs gathered from four popular AI frameworks. The research categorizes these tools’ capabilities, highlighting their strengths and weaknesses in detecting bugs. The study reveals that Flawfinder and RATS are the most effective static checkers for finding bugs in AI frameworks. However, the overall findings show that the tools detect only a negligible amount of bugs, accounting for 5/410 (0.01%) of known bugs. Based on these observations, the chapter also identifies and discusses opportunities to make the tools more effective and practical.

Chapter 4

Benchmarking Deep Learning Fuzzers

4.1 Introduction

In recent years, fuzz testing [36, 37, 38, 39, 40, 41, 42, 43], which detects bugs via generating malformed inputs, has been adopted to test DL libraries [44, 45, 46]. For example, FreeFuzz [44] is the first API-level DL fuzzer proposed to generate random inputs with various mutation strategies. Later, Xie et al. [45] proposed DocTer, which first extracts DL-specific input constraints from the documentation of DL APIs. Then the extracted constraints are used to generate valid and invalid inputs to test DL APIs. Along this line, DeepRel [46] extends FreeFuzz by using test inputs from one API to test other semantically related APIs. ACETest proposed by Shi et al. [171] automatically extracts input validation constraints from DL operators' code to generate valid, diverse test cases for bug detection. NablaFuzz [172] is the first fuzzing tool that uses the notion of Automatic Differentiation (AD) to expose bugs in DL libraries. TitanFuzz [47] is the first study to leverage the power of Large Language Models (LLMs) [173, 174, 175] to generate custom inputs for testing DL APIs. FuzzGPT [48] extended TitanFuzz by including unusual open-source LLM-generated programs to guide the generation of test input.

Although these tools have been demonstrated to be effective in detecting bugs in DL libraries and outperforming their prior work in finding more or different bugs, there remains a gap between the evaluated results and the tools' practical effectiveness. Specifically, previous lines of research failed to comprehensively evaluate the effectiveness of tools, the scope of bug detection, and their limitations against

bugs in the ground truth DL library [44, 45, 46, 47, 48]. This leaves an unanswered question: *How effectively and thoroughly can these DL fuzzers find bugs in practice?*

To our knowledge, there is no study in the literature to evaluate DL fuzzers for detecting real-world bugs based on an extensive ground-truth bug dataset. In this chapter, we conduct the first extensive empirical study to benchmark seven state-of-the-art (SOTA) DL fuzzers, including FreeFuzz [44], DeepRel [46], NablaFuzz [172], DocTer [45], ACETest [171], TitanFuzz [47], and FuzzGPT [48], against real-world bugs from two most popular and open-source DL libraries (TensorFlow and PyTorch). Specifically, we initially mined GitHub bugs using a keyword matching approach [122, 55, 56] from TensorFlow and PyTorch repositories reported between December 2020 and August 2024. Since keyword-matching may introduce false positives, we conduct a manual analysis process to verify each collected bug and filter out non-applicable bugs, i.e., bugs that cannot be triggered by malformed inputs. For each bug, we further extract its triggering APIs, which will be the target of fuzz testing tools. Consequently, we collected 517 real-world DL bugs from 379 unique issues from PyTorch and 138 issues from TensorFlow. Next, we rigorously apply the subject DL fuzzers by following their instructions on the collected real-world bugs to evaluate their effectiveness in detecting ground-truth bugs in our benchmark dataset.

Our analysis profiles the cost, effectiveness, and efficiency of each DL fuzzer from the following three aspects: the total CPU hours it takes to run (i.e., cost), the number of real-world bugs that it detects (i.e., effectiveness), and the average number of test cases it generates (i.e., efficiency). Finally, we present a detailed quantitative and qualitative analysis of bugs detected by each DL fuzzer, and we further check these fuzzers’ implementations to identify the challenges they face and the factors that affect their bug detection abilities. In this work, we investigate the following research questions:

RQ1: *What are the characteristics of extracted real-world DL bugs?*

RQ2: *How effectively and thoroughly can SOTA DL fuzzers detect real-world DL bugs?*

RQ3: *What are the characteristics of detected DL bugs?*

RQ4: *What are the reasons for fuzzers to miss a bug?*

Overall, we find that these fuzzers did not detect a large number of real-world bugs in our benchmark dataset, that is, they only detected 6.5% (34 out of 517 bugs). We observe that *Check Failed* and *Runtime Error* are the most frequent bug types in PyTorch and TensorFlow libraries. Additionally, we find that ACETest demonstrates superior bug detection effectiveness along all fuzzers. Finally, we discuss implications that can improve DL fuzzers by analyzing the root cause of bugs that cannot be

detected by these fuzzers. In general, this work complements previous studies on DL fuzzers with an extensive evaluation and provides a reference for future DL fuzzing studies. this chapter makes the following contributions:

- We conduct the first empirical study to evaluate state-of-the-art DL fuzzers against real-world DL bugs, providing a complementary perspective on prior studies regarding the comparison among DL fuzzers.
- We create the first benchmarking dataset for evaluating DL fuzzers rigorously, which includes 517 reproducible bugs from TensorFlow and PyTorch that are theoretically detectable by fuzz testing techniques.
- We conduct an in-depth quantitative and qualitative evaluation of DL fuzzers and present findings regarding the cost, effectiveness, and efficiency of the subject DL fuzzers.
- We release the dataset and source code of our experiments to help other researchers replicate and extend our study³⁴.

The remainder of this chapter is organized as follows. Section 4.2 describes the methodology of our approach. Section 4.3 and Section 4.4 show the experimental setup and the evaluation results, respectively. Section 4.5 discusses the implications for improving the subject DL fuzzers. Section 4.6 surveys the related studies. Section 4.7 presents the threats to the validity of this work. Section 4.8 concludes this chapter.

4.2 Approach

In this section, we present our approach, shown in Figure 6.3, to constructing a benchmark dataset and evaluating the effectiveness of DL fuzzers. We first explain the process of selecting DL fuzzers for our benchmark analysis (Section 4.2.1). We then discuss the selection of subject DL libraries (Section 4.2.2). Subsequently, we discuss the data collection and filtering steps, including automatic filtering and manual analysis (Section 4.2.3).

³⁴<https://github.com/dmc1778/Benchmarking-DL-Fuzzers>

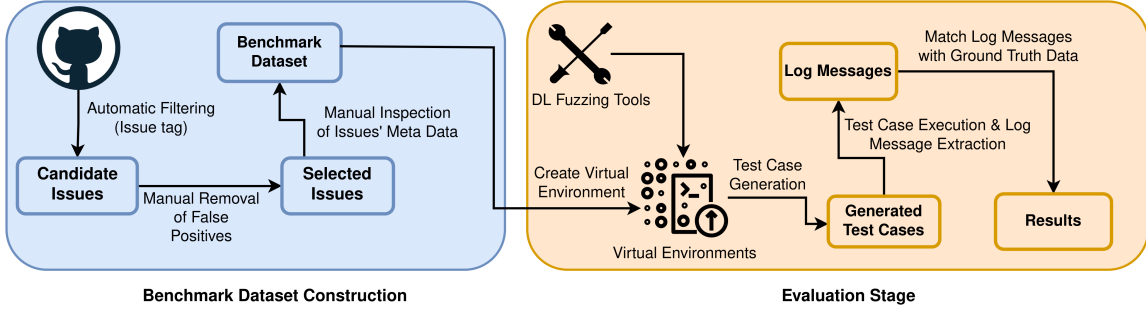


Figure 4.1: Overview of dataset construction and evaluation of DL fuzzers.

4.2.1 Selection of DL fuzzers

The focus of our research is to evaluate recent SOTA DL fuzzers, i.e., FreeFuzz [44], DeepRel [46], NablaFuzz [172], DocTer [45], ACETest [171], TitanFuzz [47], and FuzzGPT [48], which are currently being developed and available to the public. Although multiple fuzzing tools have been proposed in both academia [176, 177, 178, 179, 180] and industry [39, 40, 181], in this chapter, we pay particular attention to fuzzers that focus on DL libraries Python client APIs. There are a couple of reasons behind this decision. First, DL libraries are commonly used via their Python APIs from the front end, while the actual computations are performed in the DL backend (which is mainly developed in C/C++). While some general fuzzers use C/C++ APIs for fuzzing, testing DL libraries in their typical usage (i.e., via Python APIs) is the most effective way to detect important bugs for users. Second, DL libraries differ significantly from traditional software systems [103, 55] primarily due to their use of tensors—a unique data structure in which all computations occur at the tensor level and their non-deterministic program behavior. We chose to focus on DL fuzzers because they can simulate the tensor data structure for fuzzing, whereas general fuzzers can only model preemptive data structures. While we acknowledge the contributions of model-level fuzzers such as Muffin [182], the focus of this chapter is on benchmarking fuzzers that operate at the API level. These fuzzers are designed to identify bugs within API calls and their associated behaviors, rather than focusing on neural architectures or model design, which is the focus of model-level fuzzers like Muffin. Including both would extend the scope beyond the targeted domain of our current research. However, we have mentioned model-level fuzzing as a related but distinct area of research in the related work section of the chapter. Please note that we also exclude DL compiler testing tools [183, 184, 185, 186] since they focus on completely different subject programs, that is, they test DL compilers such as

TVM [78], TensorRT [76], and ONNXRuntime [77]. In contrast, our emphasis is on DL libraries, such as TensorFlow and PyTorch.

Table 4.1: Characteristics of DL fuzzers used in this chapter.

Fuzzer	Mutation Strategy						Test Oracles				
	Boundary-input	Shape mutation	Value/Type mutation	Prefix	Suffix	Method	Crash	CPU/GPU	Performance	Relational API	Backward/Forward Differentiation
FreeFuzz	Random	✓	✓	×	×	×	✓	✓	✓	×	×
DeepRel	Random	✓	✓	×	×	×	✓	×	×	✓	×
NablaFuzz	Random	✓	✓	×	×	×	✓	✓	×	×	✓
DocTer	Rule-based	✓	✓	×	×	×	✓	×	×	×	×
ACETest	Random	✓	✓	×	×	×	✓	✓	×	×	×
TitanFuzz	Random	✓	✓	✓	✓	✓	✓	✓	×	✓	×
FuzzGPT	History-Based	✓	✓	×	×	×	✓	✓	×	✓	✓

Table 4.1 outlines the key characteristics of the three fuzzers analyzed in this chapter. We performed a manual analysis of the implementation and documentation of each fuzzing tool to assess its capabilities and limitations. We characterize DL fuzzers mainly based on their mutation strategies and types of test oracles. This comprehensive analysis allows us to compare the effectiveness of each fuzzer.

FreeFuzz [44] is an API-level fuzz testing tool that performs test case generation using random value and type mutation. It also supports random boundary input values to generate test cases that find edge cases in DL libraries. FreeFuzz uses three types of oracles: CRASH, CPU/GPU, and performance.

DeepRel [46] extends FreeFuzz by using test inputs from an API to test other semantically related APIs that share similar input parameters. It hypothesizes that APIs with similar input parameters can share input specifications.

Its major difference when compared to FreeFuzz are test oracles where DeepRel compares two semantically related APIs in terms of equivalence value and equivalence status.

NablaFuzz [172] is an API-level fuzzer designed to test DL APIs both with and without Automatic Differentiation (AD), that is, AD partitions a program into a series of basic operations (primitives) possessing known derivatives, which can be utilized through the chain rule [187, 188].

NablaFuzz utilizes two AD modes for gradient calculation: reverse mode AD calculation and forward mode AD calculation. Additionally, NablaFuzz incorporates Numerical Differentiation (ND). In terms of fuzzing, NablaFuzz leverages the previous DL fuzzer, FreeFuzz [44], for the collection of input samples and fuzzing operations of test input (using the same mutation strategies).

DocTer [45] is a fuzzing tool that uses two major techniques for testing DL APIs: *DL-specific constraint extraction* and *DL-specific input generation*. Initially, DocTer extracts input constraints specific to DL libraries from API documentation. It generates rules from syntactic patterns found in API descriptions and applies them to a large number of API documents in popular DL libraries to extract their input

parameter constraints. The constraints are then used to guide its fuzzer to generate conform, violate, and boundary inputs to DL APIs.

ACETest [171] is a DL fuzzer designed to automatically extract input validation constraints for testing DL operators. ACETest identifies input validation code in DL operators, extracts relevant constraints, and generates test cases accordingly. The constraint models ACETest covers include arguments of type tensor, list, string, integer, float, and boolean. For tensor arguments, it models the data type, number of dimensions, total number of elements, and the shape of the tensor. ACETest employs a probabilistic technique to generate extreme edge cases with a probability of p . Similar to Docter, ACETest only supports crash oracle meaning it only detects bugs that break the Python interpreter.

TitanFuzz [47] is one of the most recent DL fuzzers, that harnesses the power of LLMs [189, 190] for conducting API-level fuzzing on TensorFlow and PyTorch libraries. As shown in Table 4.1, TitanFuzz goes beyond supporting shape and value/type mutation; it extends its capabilities to include mutation in additional code regions. Specifically, it performs mutations in the line preceding the API call sites (prefixes), the line following the API call site (suffixes), and even the mutation of method names (Method).

FuzzGPT [48] is the extension of TitanFuzz, it uses historical bug data collected from the open source to guide their fuzzer generator with LLMs. FuzzGPT initially extracts bug reports from DL library repositories to gather historical code snippets triggering bugs. FuzzGPT then focuses on specific DL library APIs, requiring corresponding buggy API labels for each code snippet.

4.2.2 DL Libraries Selection

In this study, we chose TensorFlow [71] and PyTorch [187] as our subjects, which have been widely used in the literature [191, 192, 18, 193], for a couple of reasons. These libraries have been widely used in various application domains in both academia and industry [81, 100, 194, 102, 103, 104, 105, 88, 106, 107]. Their widespread adoption ensures that any bugs detected and addressed in these libraries have significant implications for a large portion of the deep-learning community. Secondly, TensorFlow and PyTorch are open-source frameworks, providing transparency and accessibility to their codebases. This openness allows us to conduct comprehensive bug analysis and evaluation of the subject fuzzing tools in this chapter, as we have access to the source code, bug reports, and community discussions.

Collection of Real-World DL Bugs: This study aims to evaluate the effectiveness of seven state-of-the-art DL fuzzers in practice. To this end, it is essential to collect a

set of real-world bugs. In this chapter, we use the term *bug* in a general sense to refer to any kind of software defect, including security vulnerabilities, logical bugs, and performance bugs. We begin by collecting bugs from the TensorFlow and PyTorch GitHub repositories. GitHub repositories are reliable sources for identifying bugs, as users frequently report them through issues. In addition to collecting bugs from GitHub issues, we extracted bug records from commit messages referencing linked issues, if possible. This approach was chosen to improve the likelihood of finding more buggy APIs, making our dataset more comprehensive.

We gather GitHub issues and commits, including those labeled with bug or bug-fix as well as those without a bug label. We considered the tags that have a high chance of indicating bugs, such as: *type: bug, topic: bug fixes, module: edge cases, module: error checking, triaged, module: correctness (silent), high priority, bug, module: memory usage, module: int overflow, module: safe resize, module: NaNs and Infs, module: assert failure, module: crash, topic: bug fixes*, etc.

This approach accounts for issues and commits that may not be explicitly tagged as bugs but are actual bug cases. We then automatically analyze the issue title, body, comments, commit message, and code changes, searching for bug-related keywords based on insights from existing empirical studies [56, 55, 195, 111, 122]. Specifically, the following shows an excerpt of keywords we use for data collection ³⁵:

Security Vulnerabilities: *buffer overflow, integer overflow, cross-site scripting, remote code execution, memory leak, race condition, heap buffer overflow, null pointer dereference.* **Logical Errors:** *wrong result, unexpected output, incorrect calculation, inconsistent behavior, unexpected behavior, incorrect logic, wrong calculation, logic error.* **Performance Errors:** *slow, high CPU usage, high memory usage, poor performance, slow response time, performance bottleneck, performance optimization, resource usage.*

The reason behind using a keyword-based approach for filtering irrelevant issues is threefold. Firstly, it is an alignment with prior studies, as many existing empirical studies adopted a keyword-based approach for data collection [56, 55, 195, 111, 122]. Second, keywords are specific terms directly related to bug-related issues, improving the precision of issue retrieval [196]. Fortunately, PyTorch and TensorFlow have very strict policies for opening issues in their GitHub repositories. This increases the chance of finding more relevant issues using the keyword-based approach. Ultimately, keywords are selected based on common bug patterns or issues that developers typically report. This approach helps in capturing relevant and recurring bugs. At

³⁵Please note that we do not include all security keywords in the manuscript due to brevity. We list all keywords in the GitHub repository of this chapter.

the end of this automated process, 11,141 issues from PyTorch and 3,009 commits from TensorFlow were collected for our manual validation (shown in Table 4.2).

Please note that the reason behind the unbalanced amount of collected data is that PyTorch has more reported issues/commits than TensorFlow. Please note that CVE records were not included in our benchmark dataset. Concerning PyTorch, developers typically do not report bugs in the CVE repository³⁶. In the case of TensorFlow, although bug reports exist, we chose not to incorporate them for two main reasons. Firstly, CVE reports are not up-to-date compared to issues reported in the TensorFlow GitHub repository. This disparity is crucial, particularly given the evolving complexity of bug patterns in the TensorFlow library [197]. Ensuring a more current dataset is essential for robust testing of DL libraries. Secondly, the TensorFlow GitHub repository serves as the most suitable and accessible channel to report bugs within the TensorFlow community. It is worth noting that, frequently, CVE reports originate from issue reports within TensorFlow.

While the automatic mining of GitHub repositories proved efficient in gathering a substantial number of issues, it also led to the inclusion of numerous unrelated or non-bug-related issues and commits. To ensure the integrity of our manual analysis and focus solely on potential bugs, we chose to manually analyze the collected issues gathered from TensorFlow and PyTorch repositories. This detailed manual analysis helped us identify and validate actual bugs effectively. Please note that we exhaustively compared all reported issues in the replication packages of DL fuzzers with our collected benchmark data to check whether they match and remove any potential bias from our experiments and results. This stage resulted in 11,141 issues and 3,009 commits as shown in Table 4.2.

4.2.3 Manual Validation to Create Our Benchmark Dataset

Manual Analysis Criteria: Once we have automatically collected related records based on bug-related keywords from the TensorFlow and PyTorch GitHub repository, we perform manual analysis to filter false positive instances. The following are the different scopes of bugs that we used to guide our manual analysis:

Security Vulnerability: A security vulnerability or weakness is a flaw in DL libraries that external attackers can exploit to take control of the running DL application [198, 55, 56]. **Performance Bug:** Performance bugs [199] such as memory leak happens often during DL application development. For example, passing specific data types to Datapipe or dataset-related API can cause a memory leak issue ³⁷.

³⁶<https://github.com/pytorch/pytorch/security>

³⁷<https://github.com/pytorch/pytorch/issues/103581>

Logical Bug: A logical bug [126] in a DL library refers to an error in the DL application code that results in incorrect functionality or inconsistent output of DL APIs. For example, one API call (with the same input) has two different outputs under CPU and GPU implementation.

For each issue/commit, we examined the commit message and code change, issue title, body, comments, and linked PRs to determine whether the bug is detectable by fuzzing-based approaches, i.e., it can be triggered by malformed inputs. We exclude the following types of unrelated issues:

- Bugs require devices other than GPUs and CPU³⁸.
- Bugs that are related to building and library updates.
- Bugs in the backend, no high-level APIs are involved for triggering these bugs.
- Bugs in external libraries such as torchvision, torchaudio, transformers, etc.³⁹.

For each collected bug, we collected the following information: affected release (the release in which the bugs can be replicated), bug symptom, symptom category, how to trigger the bug from DL fuzzers perspective, the log message, and the buggy API. In our experiments, we randomly chose one affected release that had not yet been patched. Users may report numerous releases affected by the bug, but we opt to consider only one affected release per bug when running the fuzzers. The manual analysis required approximately 185 person-hours for PyTorch and 50 person-hours for TensorFlow to complete. This stage resulted in 559 records, 410 for PyTorch and 149 for TensorFlow. In the next stage, we conduct a manual analysis procedure to create taxonomies of trigger mechanisms.

Please note that we manually analyzed the reported bugs of the tools available in their replication packages. Fortunately, none of the issues or commits in our curated data overlap with those studied in other DL fuzzers, except for a couple of issues overlapping with ACETest. In our manual review of true detected bugs, we excluded these overlapping issues from the ACETest evaluation. No overlap was found for all other DL fuzzers. While the tools report hundreds of bugs, not all of them are relevant to our subject releases, specifically PyTorch 2.0.0, 2.0.1, 2.1.0, and TensorFlow 2.11.0, 2.12.0, 2.13.0, 2.14.0. Many bugs in our dataset reflect real-world challenges in DL

³⁸Kindly be informed that our machine is limited to CPU and CUDA capabilities only. Consequently, we filtered out bugs that necessitate different hardware platforms for replication, focusing on issues compatible with our available hardware configurations.

³⁹If the issue is merely from the external libraries, we filter them. However, there are some bugs due to the usage of DL APIs with external libraries, we also include them in our benchmark dataset.

application development, whereas the bugs reported by the studied DL fuzzers have a more limited scope. This is why the high number of bugs reported by the fuzzers is not necessarily a good indicator of their effectiveness in real-world bug detection. For instance, many of the fuzzers’ reported bugs are identified by calling individual APIs within test cases that have a limited input space. In contrast, the bugs in our dataset often involve more complex tensor operations, system-level API calls, and varied API usage.

Manual Analysis Procedure for Taxonomy Creation: Before starting the taxonomy creation, the authors started by thoroughly reading and understanding a representative sample of our data. Once the authors became familiar with the data, they started the taxonomy creation in two rounds. **Round 1:** Three authors reviewed PyTorch and TensorFlow and collected GitHub bugs. The authors extracted multiple pieces of information from each bug, i.e., the buggy API, affected release, CUDA version (if the issue is tested under CUDA), the impact or the symptom of the bug (log message or stack traces), whether the bug relates to tensor calculation, how the bug can be triggered, and bug description if necessary. The authors adopted an incremental approach to build the taxonomies. The authors first create a broad initial taxonomy that serves as a starting framework. The authors code a small batch of data using the preliminary categories. After coding, we review categories and sub-categories to identify overlaps, redundancies, or ambiguities, and refine them. The authors then code another small batch of data, applying the refined taxonomy. After the authors create taxonomies, they periodically step back to look at the entire taxonomy and assess whether it captures all significant insights from the data. Please note that the authors mark disagreements during the taxonomy creation. **Round 2:** In this round, authors started reviewing the taxonomies to resolve the disagreements. The disagreements were resolved by group discussions. In case of records where the disagreement could not be resolved, a senior involved offered an objective perspective and mediated the discussion. We continued these rounds until all disagreements were resolved.

4.3 Experiment Setup

4.3.1 Experiment Data

Table 4.2 shows the number of collected bugs and unique APIs in our benchmark. The total number of bugs (# Bugs) we could get after our manual analysis is 379 and 138 unique records for PyTorch and TensorFlow. Since the internal database of the

Table 4.3: Distribution of bug categories for each DL library.

Category	PyTorch	TensorFlow	Total
Vulnerability	33	47	80
Logical Bug	352	91	443
Performance Bug	25	11	36
Total	410	149	559

Table 4.4: Statistics on the number of overlapping APIs for each DL fuzzer on PyTorch and TensorFlow.

Tools	PyTorch		TensorFlow	
	# Covered API	# Analyzed APIs	# Covered API	# Analyzed APIs
FreeFuzz	1,071	114	1,902	68
DeepRel	1,071	114	1,902	68
NablaFuzz	1,071	114	1,902	68
DocTer	498	50	911	56
ACETest	485	35	872	24
TitanFuzz	1,329	158	2,215	106
FuzzGPT	1,377	158	2,309	106

fuzzers may not cover all APIs we collected, we consider overlapping APIs (shown in Table 4.4), i.e., if a buggy DL API exists in the internal database of the target fuzzer.

Table 4.2: Characteristics of our benchmark dataset. The numbers separated by / are shown GitHub issues/ and commits as two different sources in our dataset. The numbers in parentheses show the total number of unique records. The final real-world bug data are from PyTorch releases including 2.0.0, 2.0.1, 2.1.0, and TensorFlow releases including 2.11.0, 2.12.0, 2.13.0, 2.14.0.

Library	All Data	Automatic Filtering	# Real-World Bugs	# Unique APIs
PyTorch	13k ⁺ /69K	9,666/1,723	410(379)	249
TensorFlow	1.6k ⁺ /159K	1,475/1,286	149(138)	130
Total	-	11,141/3,009	559(517)	379

4.3.2 Configuration for Running DL Fuzzers

To configure the subject DL Fuzzers, i.e., FreeFuzz, DeepRel, NablaFuzz, DocTer, TitanFuzz, and FuzzGPT, we followed the provided instructions in the replication documentation of each DL fuzzer. We run the subject DL Fuzzers on a machine equipped with 30 CPU Cores, 214GB of RAM, and one RTX A6000, Ubuntu 22.04, and Python 3.9 for different releases of PyTorch and TensorFlow. The time cost of running each fuzzer is shown in Table 4.6.

We run the subject DL fuzzers on different releases of PyTorch (2.0.0, 2.0.1, 2.1.0) and TensorFlow (2.11.0, 2.12.0, 2.13.0, 2.14.0) as bugs in our benchmark dataset cover multiple versions of TensorFlow and PyTorch. We created separate conda environments for each version. Within each environment, we installed all the necessary dependencies and ran the fuzzing tools across different versions. Each tool was executed five times per version to ensure consistency and minimize any variability in results. As for the seed inputs, we did not collect our seeds. Instead, we used the seeds provided in the original replication packages of the fuzzers. We believe this approach ensures a fair comparison and allows us to assess the performance of each fuzzer based on the seeds used in their original studies.

4.3.3 Configuration of DL Fuzzers

In this chapter, we select seven state-of-the-art fuzzing tools, five conventional tools, and two LLM-based tools.

FreeFuzz [44]: We configured FreeFuzz to run 1000 times per API with 60 seconds of fuzzing budget. Regarding mutation strategies, we enabled all of the default and supported strategies including value, type, and database mutation. Additionally, all oracles including crash, cuda, and precision are enabled.

DeepRel [46]: We set the $top_k = 3$ which denotes k semantically similar target API to the source API that is under test. We also run DeepRel 1000 times for each API. We set the fuzzing budget for DeepRel to 60 seconds.

NablaFuzz [172]: We run NablaFuzz 1000 times per API and set the *device* parameter as *GPU* which considers GPU execution of DL APIs during test case generation.

DocTer [45]: We run DocTer using the default replication package provided by the authors which is available online. We run DocTer **1000** per each API. After carefully studying the original paper, we find that conforming inputs are considered the best configuration for test case generation. There is another mutation strategy which is called violating inputs, however, the number of detected bugs by conforming inputs is higher than violating inputs. We set the fuzzing budget of DocTer to 60 seconds.

ACETest [171]: We run ACETest 1000 times per each API with 60 seconds of fuzzing budget. We set *p_num* to 24, *max_api_error* to 50, *max_sample_error* to 5.

TitanFuzz [47]: To run TitanFuzz, we configured the seed selection strategy to *fitness*, the operator selection to *ts*, with a seed pool size of 10, and a batch size of 15. The timeout (fuzzing budget) was set to 60 seconds, and we initialized the random seed as 420.

FuzzGPT [48]: To run FuzzGPT, we employed the few-shot learning strategy, setting *k_shot* to 10, *max_length* to 256, *num_generation* to 100, *batch_size* to 10, and *generation_per_prompt* to 10. We configured the *temperature* to 0.8, *max_tokens* to 256, *top_p* to 0.95, with both *frequency_penalty* and *presence_penalty* set to 0.0. Additionally, the *sleep_seconds* parameter was set to 30.

4.3.4 Evaluation Metrics

In this study, we use the following metrics to measure the effectiveness of the seven DL fuzzers:

Number of Generated Test Cases: Is a typical metric to assess the effectiveness of DL fuzzers. This metric counts the number of distinct test cases generated by each fuzzer. We averaged the number of generated test cases over five iterations and different releases of PyTorch and TensorFlow.

Number of True Bugs Detected: Determines the number of real-world bugs discovered by a fuzzer. To calculate this metric, we iterated over the ground truth bugs in the benchmark dataset and cross-referenced them with the log messages generated by the fuzzers. We automated the process by matching the bug type from the ground truth data with the corresponding log messages generated for each test case. When a match was found, that log message was flagged as a potential candidate, narrowing down the number of log messages requiring manual evaluation. We conducted a final manual review for these potential log messages to ensure they were true positives. Please see Section 4.4.2 for more details on our manual analysis.

Execution Time: Given the distinct implementations and design assumptions of these tools, variations in execution times are expected. Consequently, we measure their performance in terms of CPU hours.

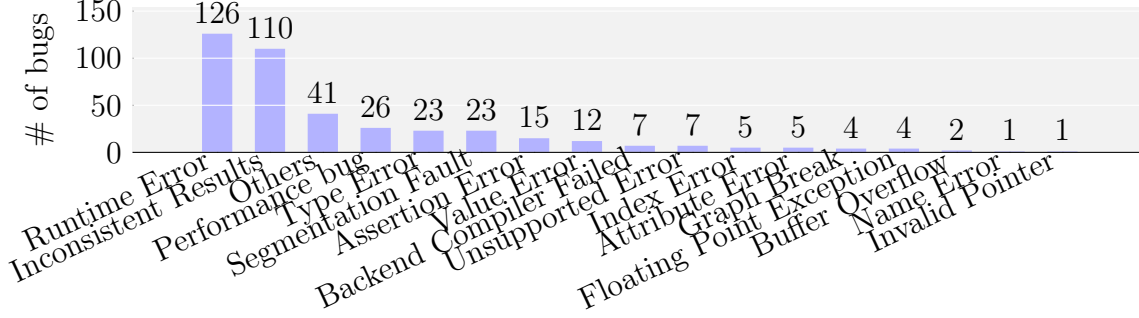


Figure 4.2: Top bug types in PyTorch library.

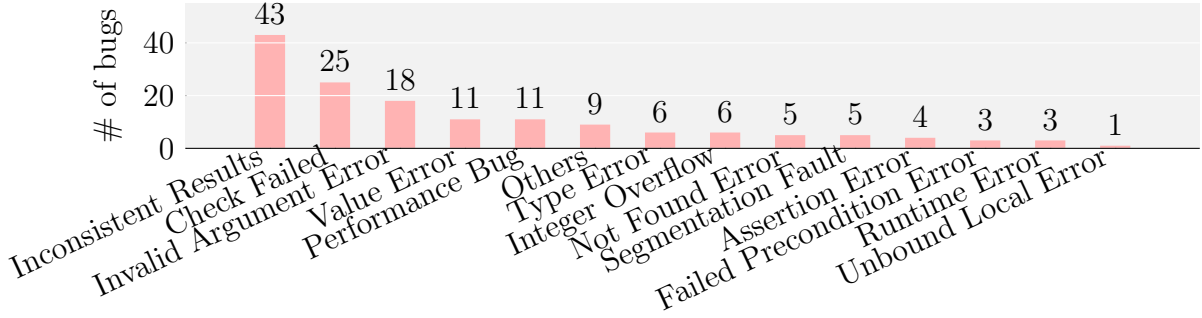


Figure 4.3: Top bug types in TensorFlow library.

4.4 Results and Analysis

4.4.1 RQ1: Distribution of Bugs Types

Approach: In this RQ, we investigate the distribution of bug types in our benchmark dataset. As the basic information of each bug has already been collected in our data collection (Section 4.2.2), we directly analyze our bug dataset for answering this RQ. We also show the number of bugs in our curated benchmark dataset reported on each release of PyTorch and TensorFlow (shown in Table 4.5).

Result: Figure 5.5 and Figure 5.6 show the distribution of bug types in the PyTorch and TensorFlow libraries, respectively. In the figures, the x-axis represents bug types, and the y-axis shows the number of issues reported for each bug type. Before we delve into analyzing the distribution of bug types, we first explain explanations for each bug type. **Runtime Error:** Occurs during the execution of a program. These could be due to issues such as memory allocation problems and invalid operations. **Inconsistent Results:** Occurs where the DL library produces incorrect outputs

based on the API reference documentation. **Performance Bug:** This Is an issue that slows down the execution of DL operations, often due to resource bottlenecks. **Type Error:** Is mainly caused by an operation on incompatible data types. For example, a function may expect a tensor but receive a different type, causing a failure. **Segmentation Fault:** Often is related to critical memory access violation where the program tries to read or write outside its allocated memory, often leading to crashes. **Assertion Error:** Occurs when a specified condition in the backend code does not hold. **Value Error:** Occurs when a function receives an argument of the correct type but an inappropriate value, such as a negative dimension size for a tensor. **Backend Compiler Faild:** Occurs due to issues in the backend compiler, which may be responsible for compiling model operations to run on specific hardware. **Unsupported Error:** Errors are caused when an operation or feature is not supported on the current hardware or configuration. **Index Error:** Occurs when accessing an invalid index in arrays or tensors, such as an out-of-bounds access attempt. **Attribute Error:** Occurs when the code tries to access an attribute or method that does not exist for an object, likely due to incorrect use of library objects. **Graph Break:** Interrupts the computational graph used in frameworks that rely on symbolic execution. **Floating Point Exception:** Is related to invalid floating-point operations, like division by zero or overflow in numerical calculations. **Buffer Overflow:** This Is a type of memory-related issue where data exceeds the allocated buffer size, potentially causing crashes. **Name Error:** Occurs due to references to undefined variables or functions. **Invalid Pointer:** Occurs where an invalid or null pointer is accessed, which can lead to segmentation faults or undefined behavior in the DL application. **Check Failed:** Occurs when a specified condition in the code is not met, resulting in the program raising an error and stopping execution. This is often due to an assert or CHECK statement in TensorFlow's C++ backend or in Python code, where specific conditions are enforced to ensure TensorFlow functions correctly. Check failures often help detect invalid states early. **Invalid Argument Error:** Occurs when a TensorFlow API receives an argument that does not meet its expected criteria (based on its reference documentation). This can include issues like mismatched tensor shapes, incompatible data types, or out-of-range values. **Integer Overflow:** Occurs when an operation attempts to create a numeric value outside the range that a data type can represent. This can lead to unintended results or errors, especially when handling large datasets, indices, or tensor shapes. **Not Found Error:** Occurs when TensorFlow cannot locate a required resource, such as a file, model, checkpoint, or operation. This error can stem from various issues, including missing files, incorrect paths, or unavailable model components. **Failed Precondition Error:** Generally indicates that an operation was attempted in an invalid state, often due

Table 4.5: Number of bugs in our benchmark datasets. The numbers inside the parenthesis show the percentage of bugs that relate to tensor calculation per release.

Library	PyTorch			TensorFlow			
	2.0.0	2.0.1	2.1.0	2.11.0	2.12.0	2.13.0	2.14.0
# of bugs	146 (75.34%)	99 (78.78%)	165 (83.63%)	26 (76.92%)	34 (73.52%)	51 (88.23%)	38 (100.00%)

to unmet prerequisites. Common causes include uninitialized variables, improper dependencies in data pipelines, or operations that require certain model components (like checkpoints) that are unavailable or unprepared. **Unbound Local Error:** Occurs when using a local variable before it has been assigned a value within the same function.

The most common bug type in PyTorch is the *Runtime Error*, with a significantly higher frequency than other categories, totaling 126 bugs. *Inconsistent Result* bugs also appear frequently, with 110 occurrences. In contrast, categories like *Name Error* and *Invalid Pointer* are much less common, each accounting for only one bug. In TensorFlow, *Inconsistent Result* and *Check Failed* are the most prevalent bug types, with 43 and 25 occurrences, respectively. The least frequent bugs are *Runtime Error* and *Unbound Local Error*, with only 3 and 1 occurrences each. The results indicate that *Inconsistent Results* is the most frequently encountered bug type across both libraries. In PyTorch, the predominant bug is the *Runtime Error*, whereas in TensorFlow, the most common issue is the *Check Failed* error. It is important to note the distinction between these two types of errors: a *Check Failed* error crashes the Python interpreter, leading to a complete termination of the program, while a *Runtime Error* does not result in a crash of the interpreter. This difference in behavior can have significant implications for debugging and error handling. Specifically, *Check Failed* errors may require more immediate attention, as they disrupt the execution flow entirely.

Table 4.5 shows the bug distribution on different releases of PyTorch and TensorFlow. We also show the number of bugs related to tensor calculation. As shown in the table, a large proportion of bugs in each release are related to tensor calculation, showing the correctness of our curated benchmark dataset as the tensor calculation is the core of DL libraries [195, 56, 200].

Answer to RQ1: In the PyTorch library, *Runtime Errors* and *Inconsistent Results* are the most common types of bugs, accounting for 126 and 110 occurrences, respectively. In contrast, in TensorFlow, *Runtime Errors* are among the least common bug types, while *Inconsistent Results* and *Check Failed* are the most frequently reported. Most bugs in our benchmark dataset are related to the tensor calculation, confirming its validity.

4.4.2 RQ2: Effectiveness of Fuzzers

Approach: RQ2 focuses on evaluating and discussing the correctness and quality of the generated test cases in connection with the ability of the studied DL fuzzers to detect real-world bugs effectively. As a result, the approaches to address RQ2 are as follows:

Step 1: Setup Execution Environments: We established multiple Anaconda environments for the targeted releases of PyTorch or TensorFlow, incorporating CUDA support and the necessary dependencies for the designated fuzzers. The installation process followed the instructions provided in the respective DL fuzzer’s manual, available within their replication package.

Step 2: Test Case Generation and Execution: With the Anaconda environments prepared, we executed DL fuzzers on each specific environment corresponding to the respective releases. The generated test cases were stored for subsequent execution. It is important to note that test case generation and execution were conducted as separate phases, primarily due to the limited computational resources of our machine. This approach ensured sufficient availability of CUDA and main memory, especially considering the substantial computational overhead associated with some of the investigated fuzzers (refer to Table 4.6). For test execution, we developed multiple Python and shell scripts to run the generated test cases in each environment (the scripts are available in the replication package). Log messages of the test case executions were captured for subsequent analysis.

Step 3: Test Case Summarization: In this step, we developed a set of scripts that automatically categorize the test cases generated by each DL fuzzer (please see Table 4.7) into one of seven categories. We provide descriptions for each category as follows: **Exception:** This status indicates that the test case encountered an error that was caught by the API under test, typically resulting in an error message but not causing the entire system to crash. For example, an API might throw a *Value Error* when given invalid input, but the program continues running. Please note that *Exception* serves as an umbrella term encompassing various categories, including

Error, *Fail*, *Not Equal*⁴⁰. We grouped these categories under *Exception* because all of them ultimately result in exceptions when test cases are run. This unified label provides a general descriptor for all such categories, ensuring compatibility across different DL fuzzers. For instance, while FreeFuzz and DeepRel support a *Fail* category, *TitanFuzz* uses *Exception*. By adopting this universal category, we standardize the test case statuses generated by each DL fuzzer. **Crash:** When a test case crashes, it crashes the Python interpreter. This usually happens when there is an unhandled error, a segmentation fault, or another severe issue that causes the program to stop unexpectedly, often requiring a restart of the system or testing environment. **Syntax:** The test case fails to run due to syntax errors, typically indicating that there was an invalid structure in the test case. Syntax errors prevent the code from running, as the language parser cannot interpret the code structure. **Out of Memory:** This status occurs when the generated test case requires more memory than is available on the system. This can be particularly common in DL tests where large data structures or tensors are involved. The Python interpreter stops or crashes due to insufficient memory allocation. **Success:** A successful test case completes without errors, exceptions, or any other issues. **Timeout:** This status happens when a test case takes longer to complete than a predefined time limit(fuzzing budget), which may indicate an inefficiency or a potential infinite loop. **Hang:** A hang is similar to a timeout but often implies that the program has become unresponsive, possibly due to a deadlock or infinite loop, and is unable to proceed. Unlike a timeout, which is based on duration, a hang may require manual intervention to stop the test case.

Step 4: Bug Detection: To detect the bug detected by each DL fuzzer, we followed a two-phase process. In the first phase, we developed an automated approach to filter test cases that are potential bugs based on our ground truth data. In the second phase, we manually investigated the filtered test cases based on a set of criteria.

In phase 1, we begin by filtering the ground truth database to include only APIs that overlap between the buggy APIs and those supported by each DL fuzzer. We then iterate through this filtered dataset and examine the log messages generated by the tool under investigation. For each entry, we compare the log message from the issue or commit description in the ground truth database with the log message generated by the DL fuzzer. If a match is found, we flag the test case as a potential

⁴⁰Not equal is specific to DeepRel and refers to cases where the output of two semantically equivalent APIs are not equal with same input values. This can refer to inconsistencies in output values—such as when both APIs are expected to produce positive outputs, but one outputs a negative value while the other is positive—or inconsistencies in status, where one API produces an exception while the other crashes.

bug candidate. Once potential bugs are identified, we conduct a detailed analysis of the log messages. Three authors jointly examine specific aspects of each case, including the type of bug indicated in the log message, the body of the log message itself, the input that triggered the bug in the issue or commits, the input generated by the fuzzer, and the environment or settings under which the bug was replicated in the issue or commit. Our automated test case filtering approach is not effective for performance bugs, as log messages provide no direct evidence of such bugs occurring. Additionally, *TitanFuzz* is the only fuzzer that supports performance bugs, specifically by detecting hangs. Therefore, we manually reviewed test cases in this category to identify those that match our ground truth performance bugs.

Result (Execution Time): Table 4.6 compares the average time costs (in CPU hours) of running the subject DL fuzzers across different releases of PyTorch (2.0.0, 2.0.1, and 2.1.0) and TensorFlow (2.11.0 to 2.14.0), based on five iterations. FreeFuzz demonstrates the lowest time cost with minimal variability, such as 0.49 ± 0.05 for PyTorch 2.0.0 and 0.31 ± 0.06 for TensorFlow 2.14.0. NablaFuzz, on the other hand, incurs the highest cost, particularly for TensorFlow 2.14.0 (10.45 ± 1.80), with significant variability. DeepRel and DocTer exhibit moderate costs, with DocTer generally performing better on TensorFlow than PyTorch. TitanFuzz and FuzzGPT show comparable time costs, with FuzzGPT slightly more efficient for TensorFlow while remaining similar for PyTorch. ACETest performs efficiently for PyTorch but incurs higher costs for TensorFlow, such as 2.17 ± 0.16 for TensorFlow 2.13.0.

The fuzzing time of each fuzzer is heavily influenced by its internal fuzzing mechanism. For example, FreeFuzz stands out as the most efficient DL fuzzer due to its straightforward mutation generation and test case execution process. It uses instrumented parameter values for each target API and randomly selects a single parameter and its corresponding values for mutation. This mutation is performed using a small set of predefined corner case values, ensuring simplicity and speed. Compared to FreeFuzz, DeepRel is the second most efficient DL fuzzer because it has to calculate the semantic similarity between APIs and executes two APIs with the same input simultaneously. Additionally, it uses a parameter K , which determines the number of semantically similar target APIs to the source/target API being tested. NablaFuzz is the least efficient DL fuzzer because it performs both backward and forward automatic differentiation for each API which requires a very high computational overhead. Although it utilizes FreeFuzz’s mutation engine, its differentiation engine significantly increases its time cost. DocTer and ACETest exhibit almost similar time costs for fuzzing because their underlying design is similar in terms of test case generation and execution. Both DocTer and ACETest perform mutation operations based on a set of DL-specific API constraints. In the realm

Table 4.6: Time cost of running each DL fuzzer averaged over 5 iterations (in terms of CPU hours).

Tools	PyTorch			TensorFlow			
	2.0.0	2.0.1	2.1.0	2.11.0	2.12.0	2.13.0	2.14.0
FreeFuzz	0.49 ± 0.05	0.50 ± 0.06	0.49 ± 0.04	0.31 ± 0.06	0.35 ± 0.03	0.32 ± 0.05	0.32 ± 0.02
DeepRel	1.50 ± 0.11	1.46 ± 0.09	1.49 ± 0.11	1.78 ± 0.39	1.89 ± 0.47	1.83 ± 0.36	1.68 ± 0.31
NablaFuzz	7.18 ± 3.32	7.17 ± 3.32	6.87 ± 3.13	6.84 ± 2.78	7.74 ± 3.55	7.69 ± 2.78	10.45 ± 1.80
DocTer	2.08 ± 0.20	2.06 ± 0.18	2.18 ± 0.06	1.35 ± 0.17	1.43 ± 0.15	1.34 ± 0.17	1.69 ± 0.09
ACETest	0.71 ± 0.02	0.72 ± 0.02	0.73 ± 0.02	2.10 ± 0.16	2.07 ± 0.05	2.17 ± 0.16	2.15 ± 0.15
TitanFuzz	5.46 ± 0.26	5.50 ± 0.29	5.60 ± 0.32	4.48 ± 0.29	4.51 ± 0.36	4.42 ± 0.31	4.66 ± 0.13
FuzzGPT	5.70 ± 0.07	5.56 ± 0.02	5.76 ± 0.01	2.62 ± 0.02	2.71 ± 0.03	2.69 ± 0.01	2.71 ± 0.007

of LLM-based DL fuzzers, TitanFuzz incurs higher time costs than traditional DL fuzzers due to its evolutionary fuzzing approach. Similarly, FuzzGPT has a high time cost because it performs multiple iterations of fuzzing operations per API.

Result (Test Case Generation): Table 4.7 presents statistics on test cases generated across various releases of PyTorch (PT) and TensorFlow (TF), averaged over five iterations. This table highlights that *FuzzGPT*, ACETest, FreeFuzz, and NablaFuzz are the leading DL fuzzers, generating results across most categories in both libraries. FreeFuzz stands out with high counts of Exception and Success cases in both PT and TF, especially in TensorFlow, where it generates nearly 500K Exceptions in PyTorch and 119,919 in TensorFlow. DocTer and ACETest also report high exception counts, particularly in TensorFlow, with 265,022 and 214,027.4 exceptions, respectively. NablaFuzz, in contrast, yields lower exception counts, with 823.4 in PyTorch and 893.8 in TensorFlow. *FuzzGPT* generates exceptions in both frameworks, though at lower levels (15,061.4 in PyTorch and 13,474.2 in TensorFlow) compared to FreeFuzz and DocTer, a trend also seen with *TitanFuzz*. In terms of crashes, NablaFuzz reports 229 in PyTorch but none in TensorFlow, while DocTer identifies 742 crashes in TensorFlow and none in PyTorch. *FuzzGPT* is unique in producing syntax errors, with 39,608.8 in PyTorch and 25,679.9 in TensorFlow. Out of Memory (OOM) cases appear only for ACETest and *FuzzGPT*, with ACETest recording 2,103 OOM cases in TensorFlow, suggesting its test generation occasionally exceeds memory limits in this library. For successful test cases, FreeFuzz leads again with the highest success rates, particularly in TensorFlow (556,916.4) and PyTorch (259,364.6). NablaFuzz has notable success counts in PyTorch (93,693.2) but much fewer in TensorFlow (1,130.8). *FuzzGPT* also achieves moderate success rates in both frameworks, with 25,238.8 in PyTorch and 12,725.05 in TensorFlow. Timeouts are seen with FreeFuzz, showing 97.4 in PyTorch and 21 in TensorFlow, while DocTer experiences more frequent timeouts, with 1,805.4 in PyTorch and 709.4 in TensorFlow.

TitanFuzz is the only tool reporting hangs, with 7.2 occurrences in PyTorch and none in TensorFlow.

Result (Bug Detection): The result analysis in this research question is based on Table 4.8 which shows the number of true bugs detected by DL fuzzers. In this table, the first column shows the overall status categories of the generated test cases. The rest of the columns show the statistics of each fuzzer in the PyTorch and TensorFlow libraries.

Table 4.7: Statistics of the generated test cases averaged on 5 iterations and different releases of PyTorch (PT) and TensorFlow (TF).

Status Categories	FreeFuzz		DeepRel		NablaFuzz		DocTer		ACETest		TitanFuzz		FuzzGPT	
	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF
Exception	490,421	119,919	45,543	26,050	823.4	893.8	232,161.8	265,022	130,963	214,027.4	7,063.6	2,408.6	15,061.4	13,474.2
Crash	25.2	30.4	-	-	229	-	-	742	464.4	883.8	-	-	2.8	61.95
Syntax	-	-	-	-	-	-	-	-	-	-	-	-	39,608.8	25,679.9
OOM	-	-	-	-	-	-	-	-	-	2,103	-	-	88.2	26.85
Success	259,364.6	556,916.4	-	-	93,693.2	1,130.8	-	-	-	-	3,404	1,455	25,238.8	12,725.05
Timeout	97.4	21	-	-	-	-	1,805.4	709.4	16.6	173	-	-	-	-
Hang	-	-	-	-	-	-	-	-	-	-	7.2	-	-	-

Table 4.8: The number of true bugs detected by DL fuzzers.

Tools	PyTorch			TensorFlow				Total
	2.0.0	2.0.1	2.1.0	2.11.0	2.12.0	2.13.0	2.14.0	
FreeFuzz	1	0	0	1	0	0	2	4
DeepRel	2	2	1	1	0	0	0	6
NablaFuzz	-	-	-	-	-	-	-	-
DocTer	0	0	0	0	1	0	0	1
ACETest	2	0	1	1	2	0	3	9
TitanFuzz	4	2	0	0	0	0	1	7
FuzzGPT	2	0	0	2	0	2	1	7

Table 4.8 demonstrates that *ACETest*, as a traditional DL fuzzer, surpasses other tools in detecting true bugs. This impressive performance, however, comes with a relatively high rate of exceptions and out-of-memory (OOM) errors compared to other DL fuzzers. Both *DeepRel* and *FreeFuzz* emerge as the second most efficient tools, performing well across multiple releases of PyTorch and TensorFlow.

FuzzGPT and *TitanFuzz* also rank as highly effective DL fuzzers in terms of bug detection. However, this commendable bug detection performance is accompanied by a relatively high rate of syntax errors for *FuzzGPT* and a higher exception rate for

Table 4.9: Performance of DL fuzzer in detecting each category of bug type.

Tool	Logical Bug	Vulnerability	Performance Bug
FreeFuzz	2	2	-
DeepRel	6	-	-
NablaFuzz	-	-	-
DocTer	-	1	-
ACETest	2	7	-
TitanFuzz	-	7	-
FuzzGPT	1	6	-
Total	11	23	-

both *TitanFuzz* and *FuzzGPT* in their test cases. This trade-off between bug detection and exception or syntactically incorrect test cases suggests that while *FuzzGPT* and *TitanFuzz* excel at identifying real-world bugs, they face challenges in generating valid test cases. *FuzzGPT* appears to employ a sophisticated strategy by guiding its fuzzer generator based on real-world bug patterns, allowing it to align test case generation closely with known bug scenarios, thus enhancing its efficiency in detecting real-world issues. In contrast, *TitanFuzz* relies on a set of in-place, location-fixed mutation operators for fuzzing. These operators guide the fuzzer generator to create code elements in predefined code locations based on the type of operator (see Table 4.1). While this approach introduces diversity into the generated test cases, it may be less effective at replicating specific real-world bug patterns. The use of broad mutation operators can result in a fuzzing process that lacks the precision and targeted focus achieved by *FuzzGPT*.

Answer to RQ2: *ACETest* excels among all subject DL fuzzers while suffering from the generation of a large number of exception test cases. Using LLM-based DL fuzzers is a *Win huge, loss is tiny* scenario where they showcase superior bug detection effectiveness with fewer exception test cases compared to traditional DL fuzzers.

4.4.3 RQ3: Characteristics of Detected Bug Types

Approach: To answer this research question, we manually verified the identified bug types by each DL fuzzer, presenting a summary in Figure 4.4 and distribution of each bug category in Table 4.9. More precisely, all three authors of this study

simultaneously examined the log messages associated with the generated test cases from each DL fuzzer across multiple releases. For each log message, the bug types are compared with those within the log messages of actual bugs present in our benchmark dataset for the corresponding releases.

Result: Table 4.9 presents the performance of the subject DL fuzzers in detecting three categories of bugs: logical bugs, vulnerabilities, and performance bugs. Among the tools evaluated, vulnerabilities were the most frequently detected, with a total of 23 instances identified across the subject DL fuzzers. Notably, ACETest and TitanFuzz demonstrated strong performance in this category, detecting 7 vulnerabilities each, followed by FuzzGPT with 6 detections. In contrast, logical bugs were detected less frequently, with a total of 11 instances. DeepRel was the most effective in identifying logical bugs, contributing 6 detections, while FreeFuzz and ACETest identified 2 each, and FuzzGPT detected only 1. No fuzzer was able to detect performance bugs, indicating a gap in current DL fuzzer capabilities.

Figure 4.4 shows the summary of bug types detected by the subject DL fuzzers. According to the depicted results, *Runtime Error* emerges as the most commonly detected bug type among the studied DL fuzzers, totaling 10 instances. Following, *Check Failed* and *Type Error* stands out as the second most frequently identified bug types, accounting for 7 and 3 instances for each bug type, respectively. *Assertion Error*, *Zero Division Error*, are the least detected bug types, each detecting only one instance.

Answer to RQ3: The most frequently identified bug type among the studied DL fuzzers is *Runtime Error*, with *Check Failed* and *Type Error* ranking as the second most commonly detected bug types. Despite *Inconsistent Result* being the second most frequent bug type in our benchmark dataset, DL fuzzers face challenges in effectively detecting this complex bug type.

4.4.4 RQ4: Root Causes for Missing Bugs

Approach: To address this RQ, we conducted a comprehensive manual examination of the implementation and documentation of the studied DL fuzzers. Our goal is to gain insight into their fuzzing capabilities and bug detection mechanisms by examining their mutation strategies and the support they provide for test oracles. Subsequently, we checked the real-world bugs overlooked by these fuzzers and developed taxonomies to categorize the underlying root causes. More specifically, our ground truth dataset consists of real-world bugs reported in GitHub issues for TensorFlow and PyTorch,

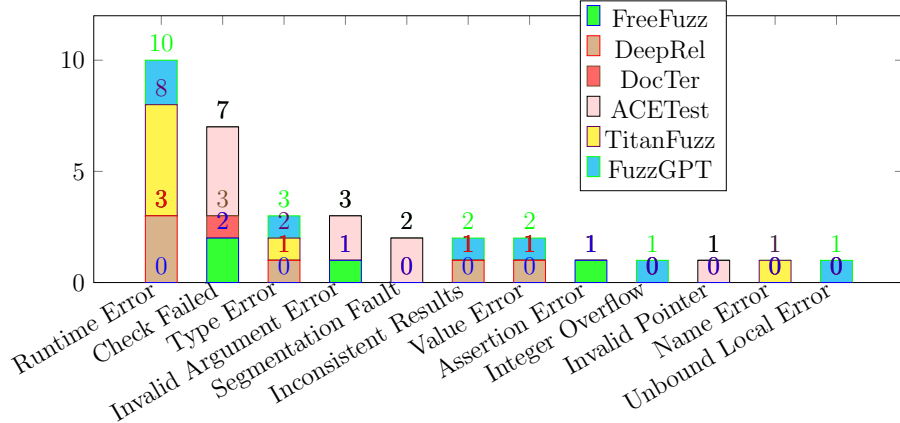


Figure 4.4: Distribution of detected bug types.

where at least one DL API is identified as the root cause of the bug. To evaluate the fuzzers, we executed each tool to determine if it could replicate the bugs described in these issues. Since the subject fuzzers do not cover every API in our dataset, we first identified the overlap between the APIs in the ground truth data and those supported by each fuzzer. We then executed the DL fuzzers on the matching APIs and marked the issues each DL fuzzer detected. For this table, we excluded the detected issues for each DL fuzzer to highlight the bugs they missed.

Result: Table 4.10 shows the summary of the root causes of the missing real-world bugs. The first two columns show the overall broad categories as well as their corresponding breakdowns. The remaining columns show the missing bugs of the studied fuzzers over each DL library. We summarize the root causes into three high-level categories, i.e., *API Input Space*, *Test Case Context*, *Test Case Context+API Input Space* with the corresponding breakdowns. For each category, we categorized the root causes which does not belong to any other category, as *Others*.

Table 4.10: Summary of root causes of missing bugs of the subject DL fuzzers. These root causes are summarised as trigger mechanisms where the fuzzers fail to detect the bugs.

Category	Bug Triggering Mechanisms	FreeFuzz		DeepRel		DocTer		ACETest		TitanFuzz		FuzzGPT	
		PT	TF	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF
API Input Space	Modeling Parameter Space	40	16	41	16	23	16	6	3	61	21	61	21
	Large Input Tensor	6	4	6	6	2	3	6	3	9	7	9	7
	Modeling Specific Data Types	16	15	16	15	9	11	6	1	19	16	19	16
	Zero Integer Argument	3	1	3	2	-	-	-	-	3	3	3	3
	Empty Input Tensor	8	1	7	1	2	2	1	1	9	6	9	6
	Large Integer Argument	2	4	1	4	-	2	-	2	2	6	2	5
	Inputs with Incompatible Shapes	5	6	5	6	1	4	1	3	4	12	5	12
	Large List Element	-	5	-	5	-	-	-	-	3	-	3	-
	Negative Input Tensor	-	1	-	1	-	2	-	-	2	-	1	-
	Negative Integer Argument	3	-	2	-	1	-	2	-	3	-	3	-
	Others	-	4	-	4	-	2	-	-	-	4	-	3
	Modeling Multiple API Usage Pattern	16	-	15	-	6	2	4	3	20	7	20	6
	Modeling Complex Tensor Operations and Layer Interactions	18	6	18	6	11	7	2	-	35	10	36	10
	Modeling Complex Tensor Operations and Layer Interactions+Modeling Multiple API Usage Pattern	5	-	5	-	2	-	-	-	7	-	7	-
Test Case Context	Model Tensor Execution on CUDA Devices	15	2	15	2	4	2	1	-	19	2	20	2
	Oracle for Performance Bug Detection	8	3	8	3	9	2	2	-	18	5	18	5
	Modeling Multiple API Usage Pattern+Modeling Compile/Eager Execution Mode	5	-	4	-	2	-	-	-	5	-	7	-
	Modeling Complex Tensor Operations and Layer Interactions+Modeling Compile/Eager Execution Mode	4	-	4	-	-	-	3	-	4	-	4	-
	Modeling Compile/Eager Execution Mode	4	5	4	4	-	5	1	5	7	11	7	11
	Model XLA Compilation	-	3	-	3	-	2	-	-	-	3	-	3
	Supporting External Library API	-	-	-	-	-	-	-	-	1	-	1	-
	Model XLA Compilation+Model Tensor Execution on CUDA Devices	-	1	-	1	-	1	-	-	1	-	1	-
	Others	1	-	1	-	-	-	-	-	2	-	2	-
	Model XLA Compilation+Modeling Parameter Space	-	1	-	1	-	1	-	-	-	1	-	1
	Modeling Complex Tensor Operations and Layer Interactions+Modeling Parameter Space	8	-	8	-	11	-	2	-	19	-	19	-
	Total	167	78	163	80	83	64	37	21	247	120	252	116
Test Case Context+API Input Space													

API Input Space: In this category, the most frequently missed bugs fall under the following types: *Modeling Parameter Space*, *Modeling Specific Data Types*, and *Inputs with Incompatible Shapes*. Figure 4.5 illustrates an example of a *Runtime Error* in the PyTorch library that went undetected by the fuzzers due to their inability to model the parameter space of the buggy API, `torch.nn.Conv2d`. As shown in the figure, the API is defined and initialized with five parameters. According to the issue description ⁴¹, setting the parameter `padding_mode` is equal to `circular` causes a *Runtime Error* due to a bug in the underlying API implementation, whereas `padding_mode` is equal to `zero` works correctly. The subject DL fuzzers struggled to replicate this bug. After analyzing FreeFuzz and DeepRel, we observed that the issue stems from their input collection process, which lacks support for the `padding_mode` parameter values necessary to replicate the error. Even if these fuzzers supported `padding_mode`, their mutation engines do not effectively explore alternative parameter values. For example, if the seed value is `padding_mode="zeros"`, the fuzzers should understand that `padding_mode` can also take `circular`, the actual trigger for the bug.

TitanFuzz, in theory, should handle parameter space modeling with its mutation operator, which supports input parameters for DL APIs. TitanFuzz uses two types of argument-handling operators: one that mutates the entire argument space to `torch.nn.Conv2d(<SPAN>)` to signal the generative model to generate seeds based on the token, and another that signals its fuzzer to add tokens as additional parameters, like `torch.nn.Conv2d(p1, p2, <SPAN>=<SPAN>)`. Therefore, theoretically,

⁴¹<https://github.com/pytorch/pytorch/issues/104860>

TitanFuzz should be capable of detecting this bug. However, it missed it, likely due to the non-deterministic nature of LLM-based tools in generating test cases. Similarly, FuzzGPT theoretically has the potential to detect this bug since it leverages historical bug knowledge to identify similar future issues. Please note that DocTer and ACETest do not support this API, so we exclude them here.

Figure 4.6 illustrates an example of a *NotFoundError* in the TensorFlow library ⁴², which was missed by the subject DL fuzzers due to their lack of support for modeling specific data types when generating input tensors. As shown, two code snippets call *tf.truncatemod* with input tensors of types *half* and *bfloat16*. The bug’s root cause is that the CPU implementation of this API does not support these particular data types. FreeFuzz and DeepRel both can model these data types, which trigger the bug in this issue, but they failed to detect the bug due to limitations in their seed selection mechanism. Using the default seed database provided in their replication packages, which we used in this chapter, we observed that the seed quality is inconsistent, preventing reliable bug replication. Upon investigating, we found that the test cases generated by FreeFuzz and DeepRel did not accurately model inputs capable of reproducing the bug, largely because of low-quality seeds. Although the *tf.truncatemod* documentation clearly states that the input should be two tensors, the seed database includes integer values as inputs, failing to meet the API’s requirements. This instability in seed quality is the primary reason for the missed detection of this bug.

DocTer is also capable of modeling the data types that trigger this issue. However, the primary reason it missed detecting this bug is due to its limited set of oracle signals. Upon examining DocTer’s implementation, we found that it only recognizes 10 types of crash signals, including *Abort*, *Bus Error*, *Floating Point Exception*, *Illegal Instruction*, *Kill*, *Broken Pipe*, *Segmentation Fault*, *NaN*, *Division by Zero*, and *System Exit*. This bug does not match any of these crash signals and does not disrupt the Python interpreter. In reviewing DocTer’s exception record list, we found the replicated log for this test case, but it was not included in DocTer’s bug list, which we consider a list of detected bugs. As a result, DocTer overlooked this bug, despite recording the relevant log message.

TitanFuzz and FuzzGPT also failed to detect this bug. Through manual investigation of the generated test cases for the buggy API across all iterations, we find that none of TitanFuzz’s and FuzzGPT’s generated test cases met the requirements for replicating this bug. Specifically, the generated test cases did not model the data types mentioned in the issue code during tensor creation. Additionally, many of the test inputs lacked specified data types and were only generated as raw tensors.

⁴²<https://github.com/tensorflow/tensorflow/issues/62070>

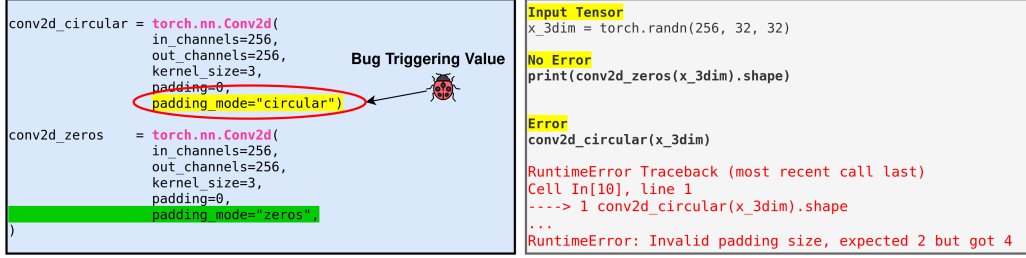


Figure 4.5: An example of Runtime Error in PyTorch library missed by the subject DL fuzzers. This bug should be triggered by *Modeling Parameter Space*.

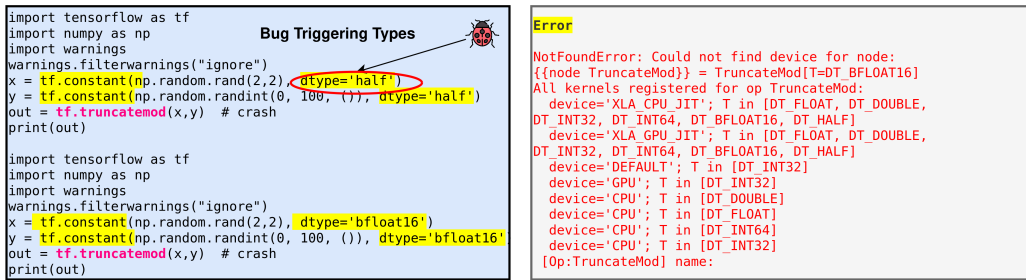


Figure 4.6: An example of NotFoundError in TensorFlow library missed by the subject DL fuzzers. This bug should be triggered by *Modeling Specific Data Types* in tensor creation.

Figure 4.7 shows an example of a *Check Failed* error in the TensorFlow library ⁴³, caused by input tensors with incompatible shapes. According to the documentation for *tf.raw_ops.BiasAdd*, the dimension of the *bias* tensor must match the last dimension of the *value* tensor. However, as shown, the code violates this condition, resulting in a *Check Failed* error that ultimately crashes the Python interpreter. TitanFuzz and FuzzGPT missed detecting this bug, despite their theoretical capability to generate test cases for this API. TitanFuzz failed to detect this bug due to its limited mutation operators; more comprehensive mutation operations are likely needed to simulate shape mismatches effectively. FuzzGPT, on the other hand, lacked the specific knowledge required to guide its fuzzer generator towards detecting mismatched shape errors.

⁴³<https://github.com/tensorflow/tensorflow/issues/62146>

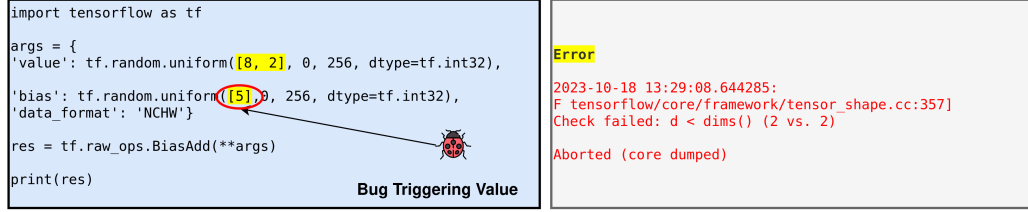


Figure 4.7: An example of Check Failed in TensorFlow library detected by ACETest and missed by TitanFuzz and FuzzGPT. This bug should be triggered by generating *Inputs with Incompatible Shapes*.

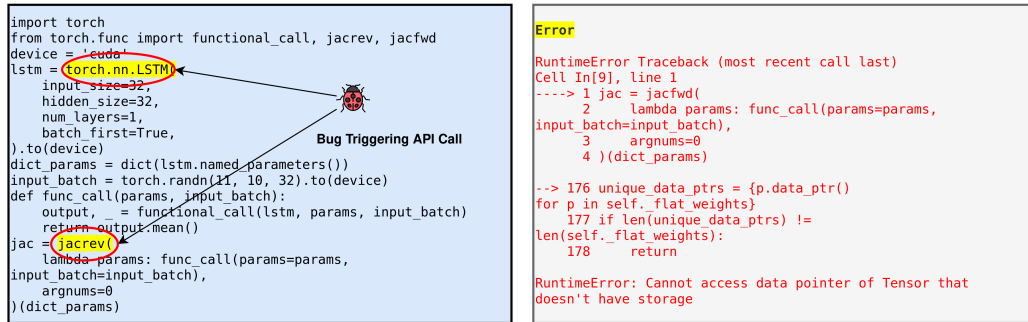


Figure 4.8: An example of Runtime Error in PyTorch library missed by the subject DL fuzzers. This bug should be triggered by calling *torch.nn.LSTM* and *torch.fun.jacrev*.

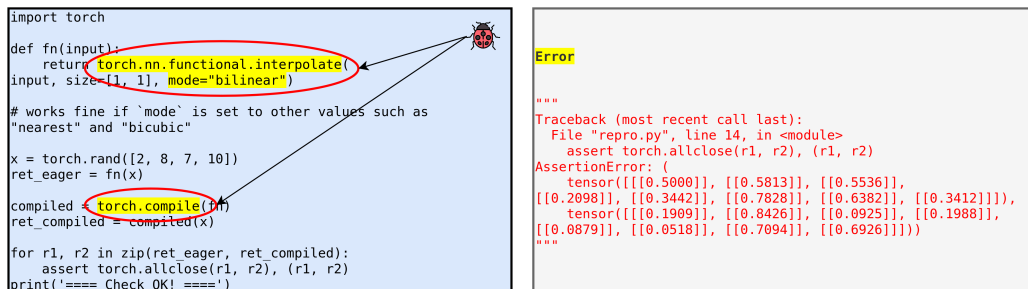


Figure 4.9: An example of Assertion Error in PyTorch library missed by the subject DL fuzzers. This bug should be triggered by calling *torch.compile* and *torch.nn.functional.interpolate* together with the optional parameter of the mode set to "bilinear".

Test Case Context: The root causes within this category pertain to various aspects encompassing the content of the test case. These include intricacies related to execution models, the representation of external libraries in the test case, configurations of

the testing environment embedded within the test case, and distinctive code patterns that can inadvertently trigger bugs.

Modeling Multiple API Usage Patterns: While creating our benchmark data, we find that certain bugs manifested as a result of invoking specific APIs consecutively or adhering to particular usage patterns involving multiple API calls [55, 200]. Figure 4.8 shows an example of bug in PyTorch library⁴⁴ where calling *torch.nn.LSTM* and *torch.func.jacrev* together raise *RuntimeError*.

Among traditional DL fuzzers, FreeFuzz, DocTer, and ACETest incorporate a single API within the generated test case, including input data. On the other hand, DeepRel and NablaFuzz attempt to model multiple API usages. However, it is worth noting that the complexity of the API usage they support is limited and falls short of capturing intricate usage patterns. Specifically, DeepRel restricts itself to incorporating semantically similar APIs using distance measures [201] (API pairs that manifest identical functionality, e.g., *torch.nn.AdaptiveAvgPool3d* and *torch.nn.AdaptiveMaxPool3d* in PyTorch). However, under this root cause category, not all API pairs have semantic similarity in terms of functionality.

LLM-based fuzzers, i.e., TitanFuzz and FuzzGPT, also suffer the same issue. Specifically, TitanFuzz utilizes mutation operators such as *Suffix* and *Prefix* to instruct the LLM to generate code elements at already-specified locations in the test case body. However, it is important to note that these operators, while versatile, do not guarantee that LLM will consistently generate API calls in the intended locations as desired. FuzzGPT is designed with the expectation of generating test cases that assess multiple usages of an API by leveraging historical data known to trigger bugs. However, the performance of FuzzGPT is heavily dependent on the quality of its constructed historical dataset. Unfortunately, FuzzGPT historical dataset construction, which guides its fuzzer generator, is performed in an ad-hoc fashion, i.e., there is no certainty as to how many bug records are specifically related to the usage of multiple APIs that are known to be buggy if called together.

Modeling Compile/Eager Execution Mode: Generally, there are two execution modes in DL libraries [202, 71], i.e., graph mode and eager mode. In graph mode, also known as JIT (Just-In-Time) Compilation, instead of executing operations eagerly, PyTorch and TensorFlow construct a computational graph that represents the operations and their dependencies. This graph is then compiled and optimized for efficient execution. Graph mode can lead to improved performance for certain workloads, especially when running on hardware accelerators like GPUs or TPUs [76, 77, 78]. In the eager mode, on the other hand, operations are executed eagerly and the results are returned immediately. It is similar to the way operations are executed

⁴⁴<https://github.com/pytorch/pytorch/issues/103161>

in imperative programming languages like Python. In certain situations, utilizing DL APIs when calling DL APIs raised exceptions under compile mode while executing normally in the eager model. A notable illustration of this inconsistency is evident in a bug within the PyTorch library ⁴⁵ shown in Figure 4.9, where calling *Tensor.copy_* and *torch.moveaxis* under compile mode raise *NameError*. None of the subject DL fuzzers support *torch.compile*, however, in the case of supporting this API, again they limit in modeling the compile mode execution of APIs both in PyTorch and TensorFlow libraries.

Test Case Context+API Input Context: In this category, multiple bug-triggering mechanisms are involved as the root cause of missing bugs.

Modeling Compile/Eager Execution Mode+Modeling Parameter Space:

Figure 4.9 illustrates an example of an *Assertion Error* in the PyTorch library ⁴⁶, which was missed by the subject DL fuzzers due to their inability to call *torch.nn.functional.interpolate* in compile mode with the *mode* parameter set to "bilinear". All subject DL fuzzers failed to detect this bug. FreeFuzz missed it because its fuzzer generator does not recognize that calling *torch.nn.functional.interpolate* with *mode* set to "bilinear" in compile mode triggers an *Assertion Error*. DeepRel also missed it, as it pairs semantically similar APIs, while *torch.compile* and *torch.nn.functional.interpolate* do not share a semantic similarity. DocTer failed to detect this bug as well because it tests only one API call per test case. Although TitanFuzz has an intelligent fuzzer generator powered by an LLM, it did not model the API call pattern needed to raise this *Assertion Error* when calling *torch.nn.functional.interpolate* in compile mode. FuzzGPT similarly missed this pattern. ACETest does not support either API involved in this bug, so it was excluded from the analysis.

Answer to RQ4: The root causes for missing detecting bugs can be summarized into three main categories, i.e., *API Input Space*, *Test Case Context*, and *Test Case Context+API Input Space*, each with specific breakdowns.

4.5 Implications for Improving DL Fuzzers

In this section of the chapter, based on the root cause analysis of missed bugs, we propose a set of actionable guidelines to improve the studied DL fuzzers in terms of effectively detecting new bugs. Given space constraints, we only provide improvements based on the major root cause patterns.

⁴⁵<https://github.com/pytorch/pytorch/issues/95262>

⁴⁶<https://github.com/pytorch/pytorch/issues/93262>

4.5.1 Extend Test Input Exploration

Generate Extreme Corner Case Values: The examined DL fuzzers have the potential for expansion beyond the mere generation of random corner case values for various parameter types within DL APIs. Instead of solely depending on random value generation, we propose a shift towards utilizing historical knowledge sourced from open repositories to inform the process of generating corner cases. For instance, traditional DL fuzzers, as observed in this study, typically generate large input tensors or integer arguments at random. However, a more effective approach involves extending these tools to generate substantial test inputs based on historical data, specifically targeting inputs known to have triggered bugs in the past.

Increase the Accuracy of Seed Collection and Selection: Traditional DL fuzzers, like FreeFuzz and DeepRel, rely on a set of initial seeds stored in their internal databases, which are collected by instrumenting live API call objects. However, this approach fails to accurately capture the real-world usage patterns of DL APIs. Our investigation revealed that some of these seeds do not represent the actual input space of DL APIs. Improving the accuracy of seed selection is strongly recommended. This inaccuracy also leads to another issue: when these fuzzers randomly select seeds from their databases, there is a risk of choosing an invalid seed. This results in test cases that are unable to effectively trigger the intended bugs.

Improvement 2: Model Parameter Correlations: Given that a significant portion of DL bugs arise from tensor calculations (as detailed in Table 4.5), we suggest the studied DL fuzzers enhance their mutation operators to cover the intricate correlation among the input parameters within DL APIs, specifically tensors and their corresponding index arguments. This correlation modeling deliberately violates DL API input constraints, thereby effectively triggering bugs in the backend implementation of DL libraries.

4.5.2 Extend Test Case Context

Model Complex Tensor Operations and Layer Interaction: In this approach, we suggest the researchers develop DL fuzzers that handle and test a variety of neural network configurations and interactions. The fuzzer should be capable of selecting and assembling different neural network layers (nn.Linear, nn.Conv2d, nn.ReLU, etc.) and architectures dynamically. It could use a set of templates for commonly used model patterns, like feedforward, CNNs, and RNNs, and combine them based on randomized or guided strategies. **Parameterized Layer Setup:** Each layer type requires specific parameters (e.g., input/output dimensions, activation functions). The fuzzer should understand layer-specific requirements to assign compatible shapes,

ensuring each layer’s output feeds correctly into the next. For complex model types, like nested *nn.Sequential* blocks or multi-branch structures, the fuzzer should be able to compose and layer multiple blocks, simulating real-world model complexity. The fuzzer should test layers with and without gradient flow control methods like `detach()` and `requires_grad_()`, observing interactions where gradients are expected to propagate selectively. This includes checks that gradients are retained or blocked as intended between layers or branches of the model. Certain layers, like batch normalization or dropout, behave differently in training versus evaluation modes. The fuzzer should be able to alternate modes and ensure that stateful layers maintain expected behaviors across training and evaluation, even in unconventional configurations.

Mine API Usage Sequence Patterns: This approach revolves around mining frequent API usage sequence patterns from bug reports using frequent itemset or graph mining algorithms [200, 203, 204]. The resulting sequences can be used as either rules to guide traditional fuzzer generators or as prompts to guide LLM-based fuzzer generators. For example, to improve TitanFuzz, the API sequences can be used along with in-place operators to increase the precision of the generated test cases, which can guide TitanFuzz to insert bug-prone API calls specifically in locations indicated by *Prefix* and *Postfix* operators rather than randomly inserting arbitrary code statements.

Parameter Exploration: Even though the most important step is to model API usage sequences and their interactions, the fuzzers must parameterize the test inputs for API calls to cover a diverse range of scenarios. Vary input sizes, data types, and other relevant parameters to explore different paths of API usage together. In other words, it is advisable to conduct parameter exploration for both the API under test and the bug-prone APIs inserted into the generated test case, which allows for simulating various API usage sequences. Additionally, we recommend ensuring that the generated test case incorporates diverse test inputs for the involved APIs.

Environment-Aware Fuzzing: The DL fuzzing tools investigated in this study have the potential for extension to produce test cases that are environment-aware. This involves incorporating a single API with identical test input data in diverse execution models, namely, eager mode or graph mode. Additionally, these test cases must incorporate an inconsistency oracle, enabling the detection of different output values resulting from the same APIs that are being called under different APIs.

4.6 Related Work

In this section, we conduct an in-depth examination of the literature concerning fuzz testing both in traditional software and DL libraries. The focus of this chapter is on

API-level DL fuzzers, so DL compiler testing tools like NNSmith [205] or NeuRI were excluded. While these tools have found numerous bugs in frameworks like PyTorch and TensorFlow, they are designed to target different types of issues, specifically related to well-formed models in DL compilers such as TVM [78], TensorRT [76], and ONNXRuntime [77]. Our study, however, is centered on bugs at the API level within DL libraries like TensorFlow and PyTorch, which require a different testing approach. We acknowledge the value of running both compiler and API-level fuzzers, but doing so would demand significant effort and resources, which fall beyond the scope of this study due to its limitations. Running DL compiler fuzzers alongside API-level and model-level fuzzers would require extensive resources that we did not have available for this work. Thus, while we recognize the contributions of NNSmith and NeuRI, they target a different problem space than what we aim to address in our study.

4.6.1 Fuzzing Traditional Software

In recent years, benchmarking studies in software engineering have attracted significant attention to assessing the efficiency and shortcomings of various software engineering methodologies, tools, or procedures in domains including but not limited to static bug detection [33, 163, 164, 206, 207, 34], reviewer recommendation system [208], API recommendation [209, 200], and automatic test case generation [210]. Also, there have been multiple studies on benchmarking fuzz testing tools on general software [211, 212, 213, 214]. The reason behind multiple studies benchmarking fuzz testing tools is twofold. First, the evaluation of fuzzers is challenging since there is no solid metric to compare fuzzers. Crash oracles are the most straightforward technique, though they suffer from deduplication problems [215, 216]. Second, the lack of a benchmark dataset results in an unreliable comparison of fuzzers, which may introduce bias in the results. Magma [211] is one of the first steps towards benchmarking fuzzers for general software to tackle the aforementioned issues. It created a benchmark dataset that contains real bugs collected from general software to enable uniform fuzzer evaluation and comparison. Magma allows for the realistic evaluation of fuzzers against a broad set of targets and enables the collection of bug-centric performance metrics independent of the fuzzer. Metzman et al. [212] introduced FuzzBench as a free, open-source platform for fuzzer evaluation. Natella et al. [213] proposed a benchmark for stateful fuzzing of network protocols called ProFuzzBench. The benchmark consists of open-source programs that implement major network protocols and tools for automating fuzzing experiments. Bohme et al. [214] is the most recent study on benchmarking fuzz testing tools on general software. The authors evaluated 10 fuzzers on 24 targets containing 13.2 million lines of code and 268 known bugs

in total. The study focuses on investigating the code coverage as a proxy measure w.r.t the detection effectiveness of the studied fuzzers. They find that there is a correlation between coverage criteria and the bug detection capability, i.e., the higher code coverage results in detecting more bugs. They conclude that employing code coverage is reasonable as a metric for fuzz testing.

4.6.2 Fuzzing DL Models

Gandalf [217] is a state-of-the-art model-level DL fuzzer that leverages a context-free grammar for generating valid inputs and a Deep Q-Network to enhance input diversity. To ensure cross-library applicability and minimize false positives stemming from API differences, Gandalf incorporates 15 metamorphic relations. Its evaluation, conducted across multiple versions of three major DL libraries and 309 operators spanning computer vision, NLP, and speech recognition, demonstrated its effectiveness in generating diverse inputs and detecting five categories of bugs. With a low false-positive rate of only 3.1%, Gandalf identified 49 unique bugs during testing.

Hung et al. [192] introduced CRADLE, an advanced method for detecting and localizing bugs in DL software libraries. CRADLE identifies bugs through two key processes: (1) performing cross-implementation inconsistency checks to detect discrepancies and (2) applying anomaly propagation tracking to pinpoint the specific library functions causing these issues. The approach was evaluated on three DL libraries (TensorFlow, CNTK, and Theano), using 11 datasets (such as ImageNet, MNIST, and KGS Go) and 30 pre-trained models. The results showed that CRADLE effectively identified 12 bugs and 104 unique inconsistencies, while successfully tracing the functions responsible for these inconsistencies.

Wang et al. [218] proposed EAGLE, an innovative approach that leverages differential testing in a novel way by employing equivalent graphs to evaluate a single DL library. These equivalent graphs are constructed using variations in APIs, data types, or optimizations that maintain the same functionality. The core idea is that executing two equivalent graphs on the same DL library with identical inputs should yield identical outputs. EAGLE introduces 16 new DL equivalence rules to generate pairs of equivalent graphs and then cross-checks their outputs to uncover inconsistency bugs within the library. When tested on TensorFlow and PyTorch, EAGLE proved highly effective, identifying 25 bugs—18 in TensorFlow and 7 in PyTorch—including 13 previously unknown issues.

Gu et al. [182] introduced Muffin, a model fuzzing technique aimed at generating diverse DL models to thoroughly test target libraries. Muffin facilitates differential testing during the model training phase by utilizing specialized metrics to detect

inconsistencies across various DL libraries, thereby increasing the chances of uncovering bugs in the library code. In evaluations on three widely used DL libraries—TensorFlow, CNTK, and Theano—Muffin effectively identified 39 new bugs in their latest releases.

Guo et al. [219] proposed Audee, an innovative testing approach for DL libraries designed to uncover various types of bugs. Audee uses a search-based method with three mutation strategies to generate diverse test cases by altering model structures, parameters, weights, and inputs. It detects logical bugs, crashes, and NaN errors. For logical bugs, Audee employs cross-referencing between libraries like TensorFlow and PyTorch to identify behavioral inconsistencies that may indicate underlying issues. To address NaN errors, it generates DNNs prone to extreme outputs, which often lead to NaNs. Furthermore, Audee utilizes causal testing to identify specific layers and parameters responsible for inconsistencies or errors. Evaluated on four DL libraries (TensorFlow, PyTorch, CNTK, and Theano) and 25 popular APIs, Audee proved effective in detecting inconsistencies, crashes, and NaN errors.

Wang et al. [220] introduced LEMON, a novel approach for testing DL libraries. LEMON consists of two main components: (1) mutation rules designed to modify DL models, enabling exploration of diverse library invocation sequences and uncovering hard-to-detect behaviors, and (2) a heuristic strategy to guide model generation, focusing on amplifying inconsistencies between DL libraries caused by bugs while minimizing noise from uncertain factors. An empirical study evaluating LEMON across 20 release versions of four popular DL libraries—TensorFlow, Theano, CNTK, and MXNet—demonstrated its effectiveness. LEMON detected 24 new bugs in the latest library versions, with seven confirmed and one resolved by developers. The study also validated the heuristic strategy’s success in enhancing LEMON’s ability to expose inconsistencies linked to detected bugs.

4.6.3 Fuzzing DL Libraries

FreeFuzz [44] represents a pioneering effort in API-level fuzz testing for DL libraries. It introduces an innovative method that utilizes open-source resources to generate test inputs. By collecting code samples from library documentation, developer tests, and public DL models, FreeFuzz instruments these samples to dynamically trace parameter types, values, and tensor shapes, enabling effective fuzz testing of various APIs. In evaluations on PyTorch and TensorFlow, FreeFuzz successfully traced valid dynamic information for fuzzing 1,158 APIs, outperforming the state-of-the-art LEMON while maintaining significantly lower overhead. During testing, FreeFuzz identified 49 new bugs across the libraries, 38 of which were confirmed by developers.

Building on FreeFuzz, its authors later introduced DeepRel [46], an advanced approach designed to expand fuzzing coverage by automatically identifying relational APIs. The key insight is that many APIs within DL libraries share similar input and output structures, allowing test inputs to be "borrowed" across APIs, thereby enhancing the testing scope. DeepRel employs two bug detection oracles—value equivalence and status equivalence—to uncover inconsistencies and identify bugs within these relational APIs. This innovative strategy significantly broadens the reach and effectiveness of fuzz testing in DL libraries.

Continuing this trajectory, the authors introduced NablaFuzz (Δ Fuzz) [172], the first fuzzer specifically designed to target Automatic Differentiation (AD) components in DL libraries. The core idea of Δ Fuzz is to abstract DL library APIs as functions operating on tensors and vectors, enabling differential testing for both output and gradient computations across various AD implementations. This abstraction allows Δ Fuzz to manage both first- and higher-order gradients, improving bug detection accuracy by filtering out false positives caused by numerical instability. Implemented as an automated API-level fuzzer, Δ Fuzz evaluates AD across multiple scenarios in major DL libraries, including **PyTorch**, **TensorFlow**, **JAX**, and **OneFlow**. It outperforms existing fuzzers in code coverage and bug detection, uncovering 173 bugs, with 144 confirmed by developers—117 of which were previously unknown and 107 directly related to AD. Over a two-month evaluation, Δ Fuzz detected 58.3% (7 out of 12) of all high-priority AD bugs in PyTorch and JAX, showcasing its superior capability in identifying critical AD-related issues overlooked by previous tools.

Xie et al. [45] proposed DocTer, a novel approach for analyzing API documentation to extract DL-specific input constraints for API functions. At its core, DocTer employs an innovative algorithm that generates extraction rules by analyzing syntactic patterns represented as dependency parse trees of API descriptions. These rules are applied to extensive documentation from popular DL libraries to systematically identify input parameter constraints. The evaluation of DocTer on three major DL libraries—TensorFlow, PyTorch, and MXNet—demonstrated its effectiveness, achieving 85.4% precision in constraint extraction. Furthermore, DocTer detected 94 bugs across 174 API functions, including previously unknown bugs that have been added to the CVE database.

ACETest [171] is a DL fuzzer that focuses on input constraints, automatically extracting input validation constraints from code to create valid and diverse test cases. These test cases are designed to effectively uncover bugs in the core logic of DL operators. ACETest works by identifying input validation code within DL operators, extracting the relevant constraints, and then generating test cases based on these constraints. Experimental evaluations on popular DL libraries—TensorFlow

and PyTorch—demonstrated that ACETest outperforms state-of-the-art (SOTA) techniques in constraint extraction quality. Specifically, ACETest extracts 96.4% more constraints and identifies between 1.95 to 55 times more bugs compared to existing methods.

IvySyn [221] is a recently introduced DL fuzzer that focuses on automatic test case generation by leveraging the statically typed nature of the backend implementation in TensorFlow and PyTorch, which are written in C/C++. The core hypothesis behind IvySyn is that the static typing in C/C++ enables type-aware fuzzing, making it more efficient compared to the computational overhead involved in capturing DL types while fuzzing Python client APIs. IvySyn uses this static typing to create a set of proof-of-concept mutation operations, which it then applies during the test case generation process. However, for the evaluation in this chapter, IvySyn was excluded from consideration. This decision was based on its heterogeneous instrumentation and fuzzing approach, which primarily focuses on functions implemented in C++, whereas the DL fuzzers examined in this study perform fuzzing directly on the front-end APIs.

Conventional fuzzing methods face challenges in the domain of deep learning testing due to the need for input programs to comply with both the syntax of the programming language (e.g., Python) and the specific constraints of DL APIs, such as tensor input shapes. To address these challenges, LLM-based DL fuzzers have been proposed. One such example is TitanFuzz, introduced by Deng et al. [47], which is the first approach to leverage LLMs for generating input programs that can fuzz DL libraries. LLMs, trained on vast datasets of code, are capable of generating realistic code snippets in an auto-regressive manner. Moreover, modern LLMs are trained on extensive DL library API usage, enabling them to implicitly learn both the syntax and specific constraints of DL APIs. TitanFuzz makes use of generative and infilling LLMs, such as Codex and InCoder, to produce diverse and valid DL program inputs for fuzz testing. The evaluation of TitanFuzz demonstrated that it achieved significantly higher code coverage—30.38% for TensorFlow and 50.84% for PyTorch—compared to existing fuzzing tools.

TitanFuzz demonstrated the ability of LLMs to implicitly learn both the syntax and the computational constraints of DL libraries, generating suitable programs for fuzz testing. However, LLMs tend to produce conventional programs based on patterns seen in their extensive training data, which may not fully explore edge cases critical for effective fuzzing. To address this limitation, Deng et al. [48] introduced FuzzGPT which built on the insight that historical bug-inducing programs may contain rare, valuable code patterns essential for discovering new bugs. Unlike traditional methods, which require manual effort to design specific generators and ensure the validity of generated programs, FuzzGPT automates this process using LLMs’ fine-tuning

and in-context learning capabilities. This approach not only automates fuzz testing but also generalizes across various domains. While FuzzGPT can be applied to different LLMs, the chapter focuses on using Codex and CodeGen, and explores the potential of ChatGPT’s instruction-following abilities to enhance fuzz testing further. The experimental study conducted on two major DL libraries—PyTorch and TensorFlow—demonstrated the effectiveness of FuzzGPT in bug detection.

4.6.4 Fuzzing DL Compilers

Recently, the field of DL compilers has garnered significant attention from researchers [205, 185, 114], as high-performance computing is critical for developing DL applications. In safety-critical domains, DL models are often compiled and optimized [76, 77] for specific hardware platforms. One of the earliest methods for testing DL compilers was introduced by [114], who developed TVMFuzz as a proof-of-concept tool to verify their root cause categorization of bugs in DL compilers. TVMFuzz was built upon existing TVM test cases. A more recent contribution, NNSmith [205], is a DL compiler fuzzer that focuses on generating valid computation graph inputs. It achieves this by leveraging a set of abstract operator models and is specifically designed to be compatible with the ONNX format. Similarly, TZer [185] introduced a functional tensor compiler fuzzer that integrates coverage guidance and combines Intermediate Representation (IR) and pass mutation. TZer performs both IR and pass mutations simultaneously, allowing it to explore a broader range of program states.

4.7 Threats to validity

Internal Validity: To avoid internal validity regarding running the studied fuzzers, we followed the original papers of the studied fuzzers to set up and run the fuzzers on the PyTorch and TensorFlow libraries. To eliminate potential biases in our collected dataset, we thoroughly compared all reported bugs in the replication package of DL fuzzers with the bugs we collected and removed the overlaps. The subject fuzzers we evaluated in this study all involve randomness. To guard against randomness, we run the tools five times on each release of PyTorch and TensorFlow libraries. All of the results in the chapter are shown based on five iterations.

In our study, we did not separate undefined behavior or soundness bugs (e.g., crash-on-invalid bugs) from logical bugs, as our primary goal was to collect and analyze a broad spectrum of bugs encountered in deep-learning libraries. However, we recognize that undefined behavior or soundness bugs, often caused by user misuse, may not reflect developer logic errors and can indeed have lower severity in production

environments where models are typically well-formed. That said, we included such bugs in our analysis because they still represent potential weaknesses in DL frameworks, particularly during development or when models are still being trained or tested. In some scenarios, even user misuse leading to crash-on-invalid issues can expose edge cases that developers may not have anticipated. Nonetheless, we acknowledge that these bugs may not be as critical as those caused by logical faults in the framework itself.

The number of overlapping APIs differs for each fuzzer. The fuzzers, such as DocTer and ACETest, operate within their built-in constraints, which results in different API coverage due to their design and the limitations of constraint handling. DocTer relies on predefined constraints, limiting its scope, whereas ACETest, operates over its constraints built on top of Z3 solver for each API. On the other hand, LLM-based fuzzers such as FuzzGPT and TitanFuzz cover a wider range of APIs because they use LLMs for test generation. While they cover different API sets, we believe the evaluation remains fair for several reasons. First, we focused on the overlapping APIs tested by the examined fuzzers to ensure a common ground for comparison, thus minimizing concerns about unfairness. Second, this reflects real-world applications where fuzzers naturally support different subsets of APIs based on their design, providing practical insights into their use cases. Additionally, extracting constraints from scratch for unsupported APIs is not always feasible, and modifying the fuzzers to extend their API coverage would require significant development efforts beyond the scope of this study. Therefore, while the number of APIs tested differs, the evaluation highlights each fuzzer’s strengths within the range it’s designed to handle, ensuring a valid and meaningful comparison.

External validity: To ensure external validity, we evaluated the fuzzers on two widely used DL libraries including TensorFlow and PyTorch. Additionally, we gathered 517 real-world bugs from their GitHub repositories. Our benchmark dataset represents the most commonly used APIs and bug types which we believe is accurately representing real-world bug distribution within DL libraries. In selecting DL fuzzers, we chose widely used open-source tools, including both traditional and LLM-based fuzzers. This selection aimed to enhance the reliability and generalizability of our findings.

4.8 Conclusion

In this study, we conducted the first evaluation of state-of-the-art DL fuzzers against real-world bugs. We manually curated an extensive DL bug benchmark dataset consisting of 517 real-world DL bugs from PyTorch and TensorFlow libraries that are detectable through fuzzing. By applying seven state-of-the-art DL fuzzers, namely

FreeFuzz, DeepRel, NablaFuzz, DocTer, ACETest, TitanFuzz, and FuzzGPT, to our benchmark dataset, we observed that these fuzzers could detect only 6.5% (34 out of 517) of the unique real-world bugs. Our analysis identified two key factors that influence the effectiveness of these fuzzers in detecting real-world bugs, offering valuable insights for future improvements in fuzzer design. Overall, this work complements prior studies by providing an extensive evaluation of DL fuzzers and establishing a benchmark for future DL library fuzzing research, paving the way for more robust and effective fuzzing tools.

Chapter 5

History-Driven Fuzzing for Deep Learning Libraries

5.1 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 [222, 223, 115], healthcare [224, 225], and financial services [226]. As a consequence, there has been a growing concern about bugs in DL systems. Moreover, as DL applications become new targets for malicious attacks [8, 227], reliability and robustness become critical requirements for these applications due to their potential impact on human life [59, 228, 229, 230, 231, 232, 233, 234].

In recent years, many fuzzers have been proposed for fuzzing DL APIs [46, 44, 45, 47, 48]. Specifically, FreeFuzz [44] 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 [46] extended FreeFuzz by random mutation of semantically related APIs in terms of equivalence values and statuses. DocTer [45] performed fuzz testing via automatic extraction of constraints from the API reference documentation of three DL libraries (i.e., TensorFlow, PyTorch, and MXNet). TitanFuzz [47] leveraged the power of Large Language Models (LLMs) [173, 174, 175] to generate inputs tailored for testing DL APIs. AtlasFuzz [48] 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 bugs, they suffer from the following limitations:

Challenge 1: Unguided Input Generation. Existing DL fuzzers [44, 45, 46, 47, 48] mainly treat the parameters of DL APIs independently in API-level fuzz testing. However, during our empirical study, we find that some bugs 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 bugs. Figure 5.1 shows an example bug⁴⁷ exposed by *mismatch between dimensions of the input tensors* in earlier releases of TensorFlow. The description of the bug 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 bugs 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 bugs triggered by specific DL API argument patterns. *Our key insight on the guided input generation of test inputs is that security bugs in DL libraries often manifest when APIs encounter a mismatch between input arguments.* Figure 5.1 (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 bugs attributed to **mismatches between input arguments** from DL APIs (see Section 5.2). 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 bugs (refer to Table 5.3 for a detailed explanation of these fuzzing heuristics rules). As shown in Figure 5.1, 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 (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.

⁴⁷<https://github.com/tensorflow/tensorflow/security/advisories/GHSA-rc9w-5c64-9vqq>

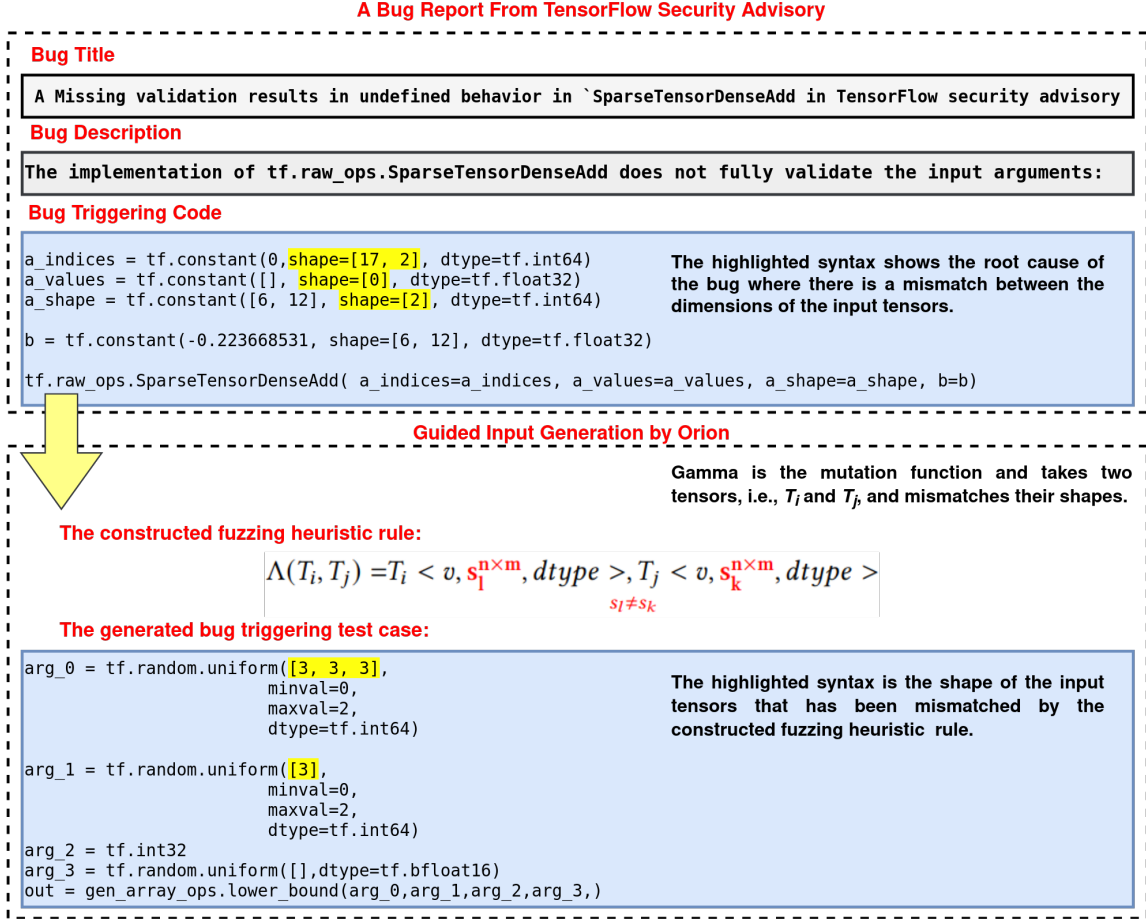


Figure 5.1: A motivational example that shows how Orion used TensorFlow security advisory to build a fuzzing heuristic rule that models the correlation between input parameters of the API `gen_array_ops.lower_bound` which eventually resulted in the detection of a segmentation fault.

Challenge 2: Lack of Support for Corner Case Generation. Existing conventional DL fuzzers at the API level [44, 46, 45, 47, 48] 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 bugs or tainted inputs external attackers provide. 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 bugs. For example, consider a documented issue within the TensorFlow GitHub

repository⁴⁸. In this case, a TensorFlow user reported a crash in the *tf.pad* API when called with `large padding sizes`. Our proposed fuzzer, *Orion*, successfully identified this bug, 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 bugs 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 bugs 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 bug data to identify fuzzing heuristic rules about the types of input, corner case conditions, or unusual scenarios that have historically resulted in software bugs. These corner case fuzzing heuristic rules (as presented in Table 5.3) 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 bugs.*

Challenge 3: Lack of Testing Developer APIs. Current conventional and LLM-based DL fuzzers [44, 46, 45, 47, 48] 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 bugs 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 bugs, 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 bugs in TensorFlow and PyTorch compared to the leading conventional DL fuzzer, that is, DeepRel. Compared to the cutting-edge

⁴⁸<https://github.com/tensorflow/tensorflow/issues/51908>

LLM-based fuzzer, i.e. AtlasFuzz, **Orion** could detect 13.63% and 18.42% more bugs in TensorFlow and PyTorch, respectively. **In terms of developer APIs**, **Orion** could outperform FreeFuzz (the only applicable baseline) by detecting 117% and 100% more bugs on TensorFlow and PyTorch respectively. This chapter 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 bug 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 advisory⁴⁹ 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.
- **Orion** can find 135 security bugs of which 76 of them are already confirmed by library developers. Among the confirmed security bugs, 69 of them are new, and 7 of them are already fixed. The rest of the reported security bugs are pending and awaiting further confirmation.

5.2 Empirical Analysis of History Bugs

To characterize and gain insight into fuzzing heuristic rules, we conducted an empirical study on 376 historical bugs 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.

5.2.1 Data Collection

In this work, we gathered historical bug data from two different sources for TensorFlow and PyTorch: issues in the GitHub repository for PyTorch and the TensorFlow security advisory⁵⁰. The reason for collecting security records from GitHub for PyTorch is

⁴⁹<https://github.com/tensorflow/tensorflow/security/advisories>.

⁵⁰<https://github.com/tensorflow/tensorflow/security/advisories>

that, at the time of writing this chapter, there were only four reported bugs in its CVE repository⁵¹. For TensorFlow, we meticulously reviewed all 407 reports available at the time of writing this chapter 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.

Automated Collection of Real-World Bugs: 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 [235, 236, 122], to automatically filter irrelevant issues and collect those related to bugs. 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 bugs:** *buffer overflow, integer overflow, integer underflow, heap buffer overflow, stack overflow, and null pointer dereference*⁵². **Logical bugs:** *wrong result, unexpected output, incorrect calculation, inconsistent behavior, unexpected behavior, incorrect logic, wrong calculation*. **Performance bugs:** *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 bug-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 bugs. 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.

Manual filtering We manually examined the collected data to obtain the final list of bugs related to PyTorch. Specifically, for each issue, we carefully reviewed the

⁵¹<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=pytorch>

⁵²To 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 chapter.

Table 5.1: 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

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:

- Bugs 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 bugs in PyTorch. In total, our root cause analysis encompassed 376 DL bug records, with 278 records sourced from TensorFlow and 98 records obtained from PyTorch.

Table 5.1 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.

Table 5.2: The notations we used in this chapter.

Notation	Explanation
Λ	A mutation function.
$T_j < v, s_{i,j}^{D_{1,2,\dots,r}}, dtype >$	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.
$s^{D_{1,2,\dots,r}}$	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 $i, j, r \in \mathbb{Z} \wedge \neq \mathbb{C} \mathbb{R}$.
$case_x$	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 <i>None</i> 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.

5.2.2 Construction of Fuzzing Heuristics Rules

To construct fuzzing heuristic rules, we initially performed manual analyses of the bug records collected to explore potential input patterns that contribute to these bugs. For CVE records, we first thoroughly examined the description of reported bugs and the provided links to the TensorFlow security advisory. Subsequently, we reviewed the bug 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 bugs. Specifically, we extracted the following factors from each record:

Root Cause: This factor represents the root cause of bugs in DL libraries, which is crucial when constructing fuzzing heuristic rules. We extracted 33 unique root cause categories⁵³ from 376 security records for both TensorFlow and PyTorch.

Reproducing Example: We checked for the presence of stand-alone code examples to reproduce the bug. Reproducing examples helps us understand which input

⁵³Due to space limitations, the details of the summarized root causes are available in the supplementary documentation at [237]

Table 5.3: 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	$\Lambda(T_1, T_2, \dots, T_j) = T_1 < v, s_1^{D_{1,2,\dots,r}}, \text{dtype} >, \dots, T_j < v, s_k^{D_{1,2,\dots,r}}, \text{dtype} >$ $s_1^{D_{1,2,\dots,r}} \neq s_k^{D_{1,2,\dots,r}}$	15	70
2 Tensors Dimension Mismatch	$\Lambda(T_i, \text{arg}_i) = \{T_i < v, s^{D_{1,2,\dots,r}}, \text{dtype} >, \text{arg}_i\}$ $\text{arg}_i \neq s^{D_{1,2,\dots,r}}$	2	5
3 Tensors List-Indices Mismatch	$\Lambda(T_i, L_i) = T_i < v, s^{D_{1,2,\dots,r}}, \text{dtype} >, L_i = \{k_1, k_2, \dots, k_j\}$ $L_i \neq D_{1,2,\dots,r}$	0	9
4 List Indices Elements Mismatch	$\Lambda(T_i, L_i) = T_i < v, s^{D_{1,2,\dots,r}}, \text{dtype} >, L_i = \{k_1, k_2, \dots, k_j\}$ $L_{ij} \neq r$	0	3
5 List Indices Length Mismatch	$\Lambda(L_i, L_j) = L_i = \{x_1, x_2, \dots, x_n\}, L_j = \{a_1, a_2, \dots, a_k\}$ $L_i \neq L_j$	0	10
Corner Case Input Generation Rules			
6 Tensor Corner Case Generator Type 1	$\Lambda(T_i) = T_i < \{\text{case}_x \text{case}_n \text{case}_{mt} \text{case}_{nan}\}, s^{D_{1,2,\dots,r}}, \text{dtype} >$	37	70
7 Tensor Corner Case Generator Type 2	$\Lambda(T_i) = T_i < v, s^{D_{\text{case}_x, \text{case}_n, \text{case}_{mt}}}, \text{dtype} >$	6	4
8 Scalar Tensor Corner Case Generator	$\Lambda(T_i) = T_i < v, s^{D_{1,2,\dots,r}}, \text{dtype} >$ $r=1$	2	8
9 Non-Scalar Tensor Corner Case Generator	$\Lambda(T_i^{s=1}) = T_i < v, s^{D_{1,2,\dots,r}}, \text{dtype} >$	0	22
10 Preemptive Corner Case Generator Type 1	$\Lambda(\text{arg}_i) = \text{case}_x \text{case}_n \text{case}_{nan} \text{case}_{none} \text{case}_{mt} \text{case}_{noa}$ $\text{arg}_i \in \mathbb{Z} \mathbb{R}$	19	22
11 Preemptive Corner Case Generator Type 2	$\Lambda(\text{arg}_i) = \{\text{case}_x \text{case}_n \text{case}_{nan} \text{case}_{none} \text{case}_{mt} \text{case}_{noa}\}$ $\text{arg}_i \in \text{String} \text{arg}_i \in \text{Boolean}$	1	10
12 List Corner Case Generator	$\Lambda(L_i) = L_i = \{k_1, k_2, \dots, k_j\}$ $k_j \in \{\text{case}_x \text{case}_n \text{case}_{nan} \text{case}_{none} \text{case}_{mt} \text{case}_{noa}\}$	10	37
13 Mutate Tensor Data Type	$\Lambda(T_i) = T_j < v, s_{n \times m}, \text{dtype} >$	6	8
Overall		98	278

specifications to DL APIs trigger bugs, aiding in the implementation of fuzzing heuristic rules.

Buggy Parameter and its Type: We also extracted information about the buggy parameter in the API input specification and its type. Collecting the type of buggy parameter is important because each parameter type has its weaknesses in terms of fuzz testing. For instance, concerning tensors, two main buggy 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 5.3. 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 5.3, 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.

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 5.3.

Tensor Corner Case Generator Type 1: In this mutation function, the value, v in $T_j < v, s_{i \times j}^{D_{1,2,\dots,r}}, dtype >$, 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 $T_j < v, s_{i \times j}^{D_{1,2,\dots,r}}, dtype >$, 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 5.3.

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 5.3.

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 5.3.

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 5.3 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 5.2.

5.3 Framework

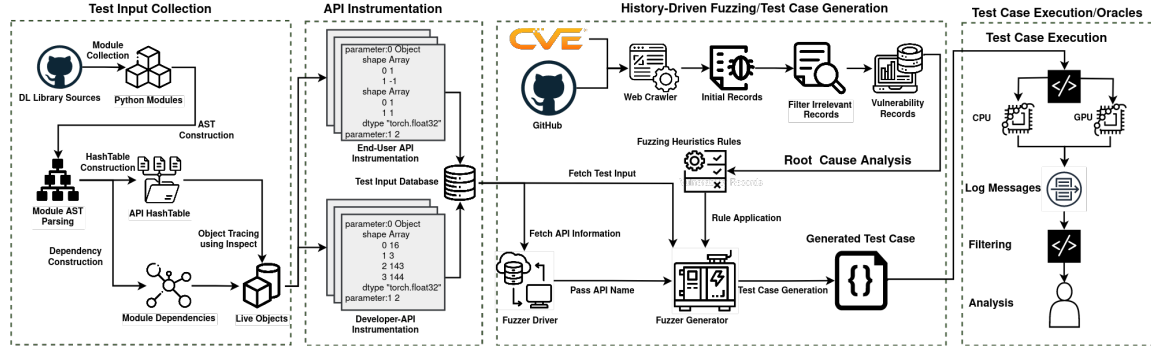


Figure 5.2: Overview of our proposed Orion.

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 5.3.1.

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 5.3.1.

History-Driven Fuzzing/Test Case Generation (Stage 3): The third stage introduces our history-driven fuzzer, as described in subsection 5.3.2. 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 5.3.3. 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.

5.3.1 Test Input Collection and API Instrumentation

Following existing work [44, 46], **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 [44] 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 5.2, as a lightweight static analyzer leveraging the Python AST module⁵⁴. 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 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

⁵⁴<https://docs.python.org/3/library/ast.html>

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 module⁵⁵. 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 bugs in these APIs can indirectly affect the end-user experience. For example, bugs in developer APIs could lead to performance issues or bugs 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. Bugs in developer APIs could potentially be leveraged by attackers to bypass security mechanisms, compromise model integrity, or cause unintended behavior in deployed TensorFlow models.

5.3.2 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 $L_0(t), L_1(t), \dots, L_{n-1}(t)$, 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.

⁵⁵<https://docs.python.org/3/library/inspect.html>

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 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 $\text{type}(t)$, 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 for execution.

Algorithm 1 History-Driven Fuzzer Generator

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

5.3.3 Test Case Execution

In this stage, the generated test case is executed to identify potential bugs. 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 bugs are not related to crashes and are filtered out using our approach. Please note that we only use this approach to filter crash-related bugs from non-crash-related ones. This helped us a lot during bug 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.

5.4 Experimental Setup

5.4.1 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 bugs?
- **RQ3:** What is the contribution of Orion’s fuzzing heuristic rules in detecting bugs?

5.4.2 Subject DL Libraries

In this chapter, 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.

5.4.3 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.

5.4.4 Baseline Approaches

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

FreeFuzz [44]: 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 [46]: 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 [45]: 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.⁵⁶

TitanFuzz [47]: To run TitanFuzz, we configured the seed selection strategy to *fitness*, the operator selection to *ts*, with a seed pool size of *10*, and a batch size of *15*. The timeout (fuzzing budget) was set to 60 seconds, and we initialized the random seed as 420.

AtlasFuzz [48]: To run FuzzGPT, we run it with few-shot learning strategy with *k_shot* to 10, *max_length* to 256, *num_generation* to 100, *batch_size* to 10, *generation_per_prompt* to 10.

In this chapter, we directly use the reported bugs 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 documentation⁵⁷. 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 history⁵⁸. 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.

5.4.5 Evaluation Criteria

Following existing work [45, 44], 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 bugs will be discovered and fixed.

Number of Bugs 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, FreeFuzz, Docter, and DeepRel. Specifically, we use detection rate and fixed rate as two metrics for the

⁵⁶Please note that there is another setting called violating input, though, the number of detected bugs by Conforming inputs is higher than violating inputs.

⁵⁷<https://www.tensorflow.org/install/source>

⁵⁸<https://pytorch.org/get-started/previous-versions/>

comparison. The detection rate indicates the detection performance of the fuzzer while the fixed rate indicates how many bugs 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 Bugs: The primary objective of this study is to uncover new bugs in the latest releases of TensorFlow and PyTorch, thus we also report the number of new bugs 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 chapter to the detection of new bugs in the latest releases of TensorFlow and PyTorch.

5.4.6 Result Analysis

5.4.7 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.

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 manual⁵⁹ 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 directory⁶⁰.

RQ1.2: Comparison Against the Traditional DL Fuzzers: In comparison with the baseline fuzzers, we provide details on the number of bugs 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 chapter, 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

⁵⁹<https://www.tensorflow.org/install/source>

⁶⁰<https://github.com/pytorch/pytorch/tree/master/test>

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 bugs mentioned in their replication package. The rationale is the compatibility issue of LLM-based fuzzers with previous releases of TensorFlow and PyTorch (see Section 5.4.4). Hence, we use TensorFlow 2.10.0 and PyTorch 1.12.0 for the comparison.

Table 5.4: Statistics of covered APIs by each DL fuzzer. The percentage of improvements is computed relative to Orion.

	PyTorch		Developer APIs	TensorFlow	
	End-User APIs	% Improvement		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	-

Results: API Coverage Results: Table 5.4 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 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.

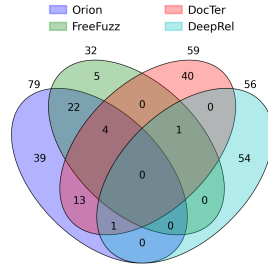
The reason for the high number of covered APIs for TitanFuzz and AtlasFuzz is that given a list of API names, directly generate sample runnable code for each end-user API by using LLMs and the sample code snippets for fuzz testing and test case generation. In contrast, in our approach, we begin by collecting 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-runable 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.

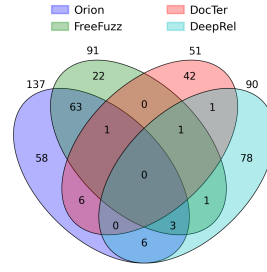
Orion vs Traditional DL Fuzzers: Table 5.5 shows the detection effectiveness of Orion compared to FreeFuzz on developer APIs. As shown, Orion outperforms FreeFuzz on both TensorFlow and PyTorch releases significantly. More specifically, Orion detects 79 and 60 more bugs versus FreeFuzz. Table 5.6 shows the detection effectiveness of Orion compared to the baseline fuzzers. It is observable that Orion excels in terms of bug 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 bugs for end-user APIs, showcasing its robustness in bug detection in TensorFlow 2.4.0. In terms of PyTorch 1.7.0, Orion outperforms all three tools with 5 more bugs 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 bug detection, but is surpassed by DeepRel.

Figures 5.3 present Venn diagrams that illustrate the number of bugs detected by both Orion and traditional baseline methods. A key observation is that Orion outperforms in discovering a greater total number of bugs across both TensorFlow and PyTorch releases. Regarding traditional DL fuzzers, Orion consistently detects the highest number of overlapping bugs when compared to FreeFuzz across all releases of TensorFlow and PyTorch. In contrast, compared to LLM-based fuzzers, Orion detects fewer overlapping bugs. This difference can be attributed to the fact that Orion focuses on bugs, whereas LLM-based fuzzers primarily identify general DL bugs.

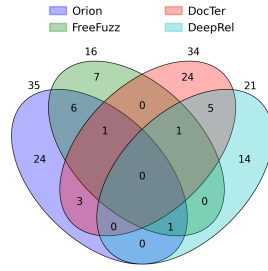
Orion vs LLM-based DL Fuzzers: Table 5.8 shows a comprehensive comparison between Orion and LLM-based DL fuzzers, namely TitanFuzz and AtlasFuzz. The results demonstrate Orion’s superior performance, particularly evident in TensorFlow 2.10.0 and PyTorch 1.12.0, where it exhibits a substantial improvement of 13.63% and 18.42%, respectively. Furthermore, Figure 5.3 provides insights into the overlap



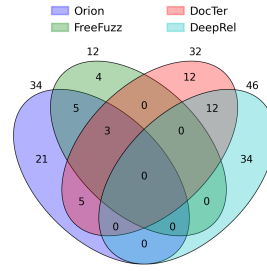
(a) TensorFlow-2.3.0



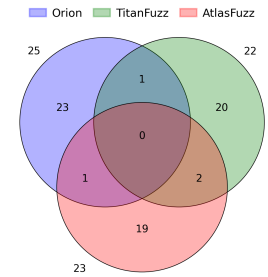
(b) TensorFlow-2.4.0



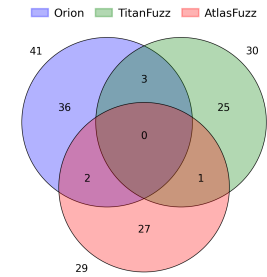
(c) PyTorch-1.7.0



(d) PyTorch-1.8.0



(e) TensorFlow-2.10.0



(f) PyTorch-1.12.0

Figure 5.3: Overlap of the number of detected bugs on TensorFlow and PyTorch releases.

in the number of detected bugs 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 bugs. Specifically, **Orion** focuses primarily on the detection of crash bugs, while TitanFuzz and AtlasFuzz focus on logical bugs. This divergence in focus results in a higher intersection of detected bugs in the case of PyTorch, as illustrated in Figure 5.3f, 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 bugs and the distinct nature of bugs 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 5.5: 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 5.6: 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

Table 5.7: 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 5.8: 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 bugs 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 bugs on TensorFlow and PyTorch releases.

5.4.8 RQ2: Performance in Detecting New and Known Bugs

Table 5.9 presents statistics on newly detected bugs by Orion in the latest releases of TensorFlow and PyTorch. In total, Orion reports 135 bugs, 76 of which have already been confirmed by library developers, and 69 of them are new bugs. Since we report the bugs on the most recent releases for each library, at the time of writing this chapter, merely seven of them have been fixed. The number of reported bugs 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 bugs within the TensorFlow library is higher since TensorFlow developers are more active in triaging the reported bugs.

Figure 5.4a illustrates an example of a segmentation fault, which is one of the bugs detected by Orion in PyTorch. This bug 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 bugs that share the same root cause.

Figure 5.4b showcases a bug that **Orion** detected in TensorFlow. This bug 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 bug.

Table 5.9: The Number of new detected bugs. 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
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,)
```

Vulnerability

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

```
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,)
```

Vulnerability

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

Figure 5.4: Example bugs detected by Orion.

Figure 5.5 and Figure 5.6 illustrate the total number of bugs 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 bugs. Following closely behind is *Segfault*, which emerges as the second most detected bug in the PyTorch library. On the other hand, in TensorFlow, *Segfault* takes the lead as the most detected bug 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 bug, with 12 reported instances.

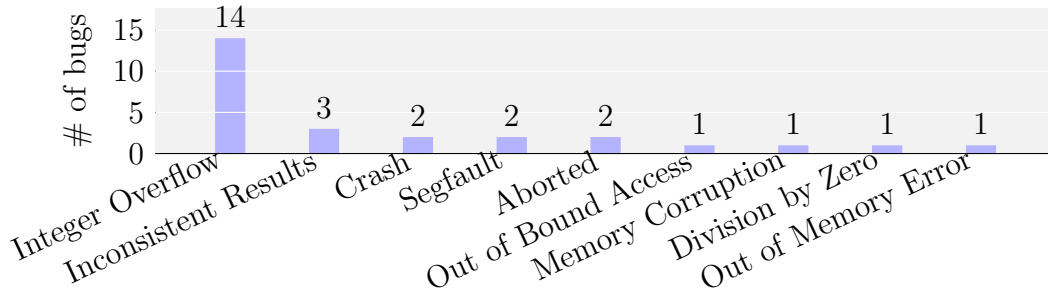


Figure 5.5: Total number of confirmed bug types detected by Orion in PyTorch library.

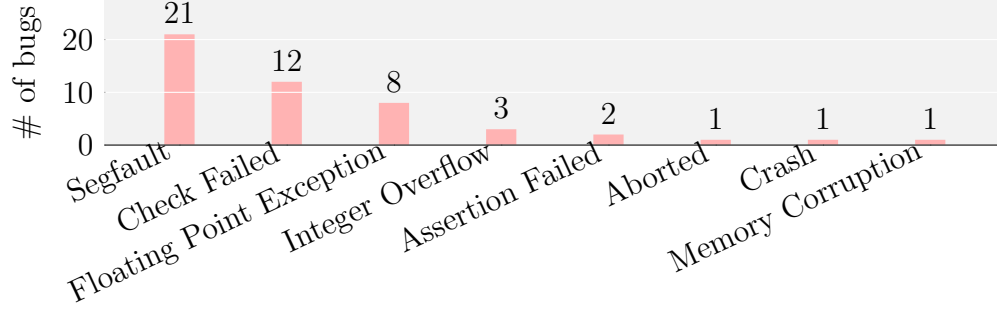
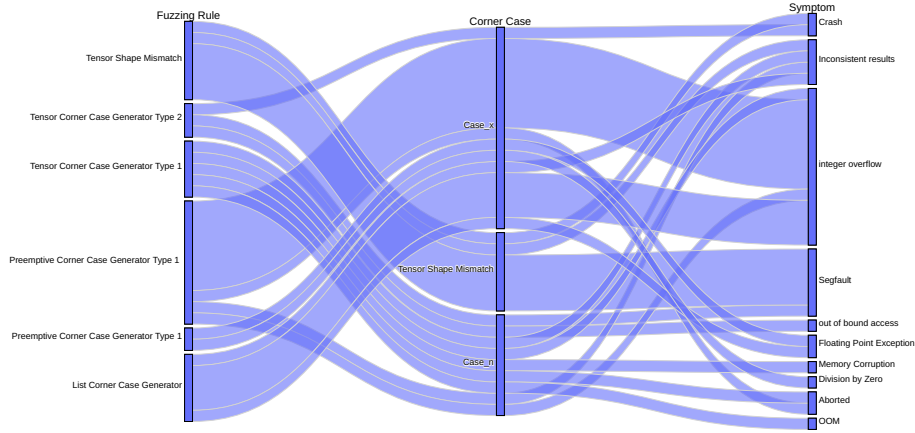


Figure 5.6: Total number of confirmed bug types detected by Orion in TensorFlow library.

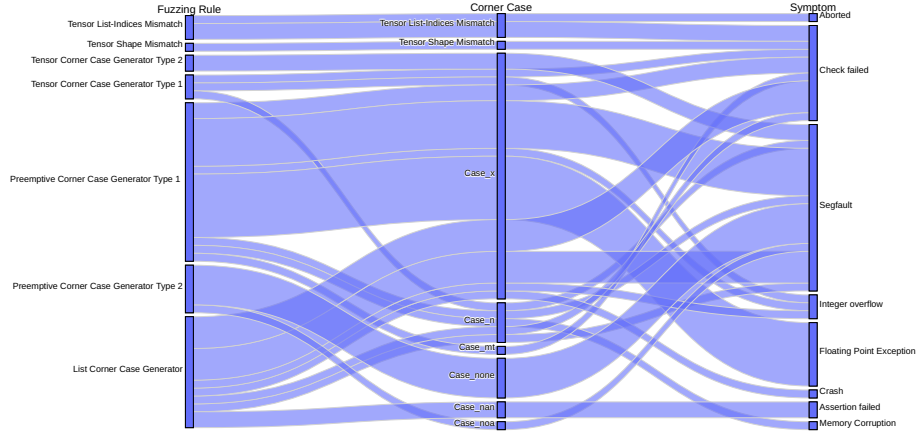
Figure 5.7 shows the mapping of root causes to the symptoms of detected bugs including new and known ones for PyTorch and TensorFlow. In the figures, the first columns show the overall fuzzing rules, while the second column specifies the specific corner cases utilized by the fuzzing rules for bug detection. The last column shows the symptoms of the detected bug.

From Figure 5.7a 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 prone 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 5.7b 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, bugs 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 bugs resulting from *case_x* tend to lead to *Segfault* and *Floating Point Exceptions*, unlike PyTorch library where *case_x* leads to *Integer Overflow*.



(a) PyTorch mapping.



(b) TensorFlow mapping.

Figure 5.7: Mapping of fuzzing heuristic rules to bug symptoms.

Answer to RQ2: Orion can detect 135 bugs, of which 76 have been confirmed by library developers. Among the confirmed bugs, 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.

5.5 RQ3: Ablation Study of Fuzzing Heuristic Rules

Table 5.10 illustrates the contributions of fuzzing heuristic rules to the detection of bugs in TensorFlow and PyTorch. Overall, the rules categorized under *Corner Case Input Generation* demonstrate a greater impact on detecting bugs 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 buggy 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 the distribution of fuzzing heuristic rules contributed to the detected bugs by Orion is consistent with the real-world distribution of root causes shown in Table 5.3. In the category of *Corner Case Input Generation Rules*, the *case_x* and *case_n* emerge as the most effective in bug 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.

Table 5.10: Contribution of Orion’s fuzzing heuristics rules on detecting bugs 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

Answer to RQ3: The *Corner Case Input Generation* category stands out as the most effective in terms of fuzzing heuristic rules for detecting bugs. 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 bugs.

5.6 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 [44, 45]. In the future, more metrics will be explored.

5.7 Related work

5.7.1 History Driven Fuzzing

History-driven fuzz testing is a testing approach [238, 239, 240, 42] that takes advantage of historical data, typically in the form of previously discovered bugs, 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. [239] 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. [240] 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. [42] 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 bugs than the baseline and achieves greater line coverage than other fuzzers in most programs. Chen et al. [238] 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.

5.7.2 Fuzzing DL Compilers

Recently, many researchers have been attracted to the research of DL compilers [205, 185, 114]. 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 [114] 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. [185] 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 [205] 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 bugs. 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 bug analysis to guide test generation. Another primary component of Orion is an empirical study in which it analyzes 376 bugs 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 bugs in TensorFlow and PyTorch; 76 were confirmed by developers, with 69 previously unknown. Compared to other tools, Orion detected significantly more bugs 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 bug 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.

5.7.3 Testing DL Libraries

Deep learning libraries have been widely used to assist users in Deep Neural Networks (DNNs) training and prediction tasks [241] such as image classification [81, 82, 83, 84, 85, 86, 87], natural language processing [88, 89, 90, 91, 92, 93, 94, 95], and software engineering [2, 96, 97, 98, 99].

Researchers have conducted numerous research studies [191, 192, 242, 223, 222, 243, 244, 245, 246, 247, 248] for testing DL libraries. One of the first steps toward testing DL libraries is the framework proposed by [192] 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 [191] 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 [45, 44, 46, 249] where each API is considered a subject for fuzzing based on a set of predefined or random mutation rules. For example, DocTer [45] designed to analyze API documentation to extract deep learning (DL)-specific input constraints for DL API functions. FreeFuzz [44] 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 [46] 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 [249] 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 [44] and DeepRel [46] adopt approaches that involve generating random test inputs for API-level testing of TensorFlow and PyTorch. DocTer [45], 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 [47] and AtlasFuzz [48], 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 vulnerabilities. In contrast, our approach is centered on the construction of fuzzing heuristic rules based on historical bugs. These rules are designed to mimic real-world corner cases test inputs when fuzzing DL APIs, thereby discovering critical bugs 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 bugs in DL libraries.

5.8 Conclusion

In this chapter, 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 bugs 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 bugs. 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 bugs, 76 of which are confirmed by the community of DL library developers. Among the 76 confirmed bugs, 69 are new bugs, 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 bugs, respectively. In comparison with the state-of-the-art LLM-based fuzzer, e.g., AtlasFuzz, **Orion** detects 13.63% and 18.42% more bugs on TensorFlow and PyTorch.

5.9 Data Availability

We share the source code of **Orion**, the bug list, and data in [237].

Chapter 6

Taming Checker Bugs in Deep Learning Libraries

6.1 Introduction

Research on automatically finding and fixing checker bugs in software systems has shown promising results in improving software reliability [49, 50, 51, 52, 53, 54]. However, while much work has been done on this topic for conventional software, little attention has been paid to similar issues in deep-learning (DL) libraries. This lack of focus is mainly because deep learning libraries are developed differently from traditional software [55, 56, 57]. As a result, most of the current related research is centered on checker bugs of conventional software, leaving a significant knowledge gap regarding bugs specific to DL libraries.

Checker bugs refer to defects arising from the absence or incorrect implementations of validation checks within software systems' code bases. These issues can manifest themselves as incorrect functionality or system crashes. In DL libraries, checker bugs have unique characteristics, as these libraries are constructed differently from traditional software [55, 56, 57]. DL libraries are built on a specialized data structure called a tensor [71], and consequently, the majority of bugs result from inadequate validation of tensor properties [55]. Fig. 6.1 illustrates an example of a PyTorch checker bug where the developer failed to verify if an unsigned integer variable k , i.e., representing an index of the input tensor, is less than the size of the input tensor's dimension. This type of oversight can lead to critical errors in tensor operations and data processing within DL systems.

```

89 inline int64_t size_between_dim_(int k,
90 int l, IntArrayRef dims) {
91     -TORCH_CHECK((unsigned)l < dims.size());
92     +TORCH_CHECK((unsigned)l < dims.size() &&
93     +(unsigned)k < dims.size());
94     ...

```

Figure 6.1: An example of a checker bug (#76715) in PyTorch library. The original checker failed to validate if the index k is less than the size of the dimension of the input tensor.

Existing studies on detecting and fixing checker bugs particularly focus on traditional software [49, 250, 50, 51, 52, 53, 54, 251, 252, 253, 254, 255, 256, 52].

Jana et al. [257] first proposed to use of error specifications to detect checker bugs in C programs. Along this line, Tian and Ray [49] comprehensively investigated real-world checker bugs in C programs and introduced ErrDoc, a tool aimed at detecting and automatically repairing these bugs.

Wang et al. [54] introduced AMCheX, an LLVM-based tool for accurate missing-check analysis in the Linux kernel. Wang et al. [253] developed a static analysis tool to identify instances where critical operations in C/C++ code lack proper validation of potentially unsafe input. Despite the prevalence of studies on checker bugs of traditional software, we identified limitations of existing studies on checker bugs.

First of all, there is a lack of a comprehensive classification of checker bugs for DL libraries. The classification of checker bugs for traditional software explored in previous studies, such as Tian and Ray [49], only focused on checker bugs related to *IF* statements. However, we find that checker bugs in DL libraries are more diverse and require a broader classification. This discrepancy motivates us to develop a more comprehensive classification of checker bugs for DL libraries.

Second, the traditional checker bug detection and repair approaches, e.g., ErrDoc [49] and AMCheX [54] only focus on validating the conditions in *IF* statements. However, in DL libraries, validation, and error-checking mechanisms are often more complex and often encapsulated within custom macros or functions (e.g., *OP_REQUIRES* and *TORCH_CHECK*). This complexity introduces unique challenges for traditional bug detection and repair tools. For example, the following code snippet shows the content of *OP_REQUIRES* which involves several steps, i.e., predicting the condition, checking computation context, handling failure, and returning. Traditional detectors may struggle to parse and understand this multi-step logic. In addition, traditional checker bug detection tools are designed based on

simple if-else change patterns. Patterns within these DL macros can include multiple function calls, context-specific operations, and detailed error-reporting mechanisms. In *OP_REQUIRES*, for instance, the macro not only checks a condition but also performs additional tasks like failure handling and context checking, which go beyond a simple condition checker.

```
define OP_REQUIRES(CTX, EXP, STATUS)
do {
    if (!TF_PREDICT_TRUE(EXP)) {
        CheckNotInComputeAsync((CTX),
            "OP_REQUIRES_ASYNC");
        (CTX)->CtxFailure(__FILE__, __LINE__,
            (STATUS));
        return;
    }
} while (0)
```

To alleviate the mentioned issues, in this chapter, we conduct the first comprehensive analysis of DL checker bugs within two major and widely used DL libraries, namely TensorFlow and PyTorch. For our analysis, we collected 527 checker bugs with a keyword-based heuristic approach (details are in Section 6.2). To understand the characteristics of checker bugs, we analyzed each bug from three perspectives, i.e. root cause of bugs, symptom, and fixing patterns.

To detect and repair these DL checker bugs, we propose **TensorGuard** as a proof-of-concept tool that leverages LLMs through prompt engineering with a RAG database. We experiment with ChatGPT⁶¹, i.e., GPT-3.5-turbo, due to its demonstrated effectiveness in software engineering tasks such as code generation [258] and program repair [259]. Specifically, for building the RAG vector database [260], we collected 211K commits from the PyTorch and TensorFlow GitHub repositories in 8 years from September 2016 to December 2023. We evaluated **TensorGuard**’s performance on a test dataset that includes 92 buggy and 135 clean checker-related changes in TensorFlow and PyTorch from January 2024 to July 2024.

We evaluated **TensorGuard** in three different prompting strategies including Chain of Thought (CoT), Zero-Shot, and Few-Shot prompting strategies. Our results demonstrate that **TensorGuard** has a high average recall (94.51%) using Chain of Thought prompting, a balanced performance between precision and recall using Zero-Shot and Few-Shot prompting strategies. In terms of patch generation, **TensorGuard** generated 90 patches for DL checker bugs in the test dataset, 10 of which were correct.

⁶¹<https://chatgpt.com/>

To detect new checker bugs, we applied **TensorGuard** on 87 checker-related commits (including 493 changes) of the JAX library from Google which resulted in the detection of 64 new checker bugs and **TensorGuard** successfully fixes four of them.

The contributions of this chapter are as follows:

- We present the first large-scale analysis to demystify checker bugs within PyTorch and TensorFlow libraries.
- We create a benchmark dataset including 527 instances of checker bugs, providing detailed taxonomies regarding the root causes, symptoms, and fixing patterns of these checker bugs.
- We introduce a novel LLM-based DL checker bug detection and repair tool, **TensorGuard**. Our evaluation shows that our tool outperforms the state-of-the-art.
- We offer practical guidelines to help the development teams of TensorFlow and PyTorch handle checkers.

The structure of the remainder of the chapter is organized as follows. Section 6.2 outlines the empirical study of DL checker bugs. Section 6.3 discusses our proposed tool and its results in DL checker bug detection and repair. Section 6.4 discusses our findings. Section 6.5 addresses potential threats to the validity of our study. Finally, Section 6.6 provides a summary of the chapter.

6.2 Characterizing Checker Bugs in DL Libraries

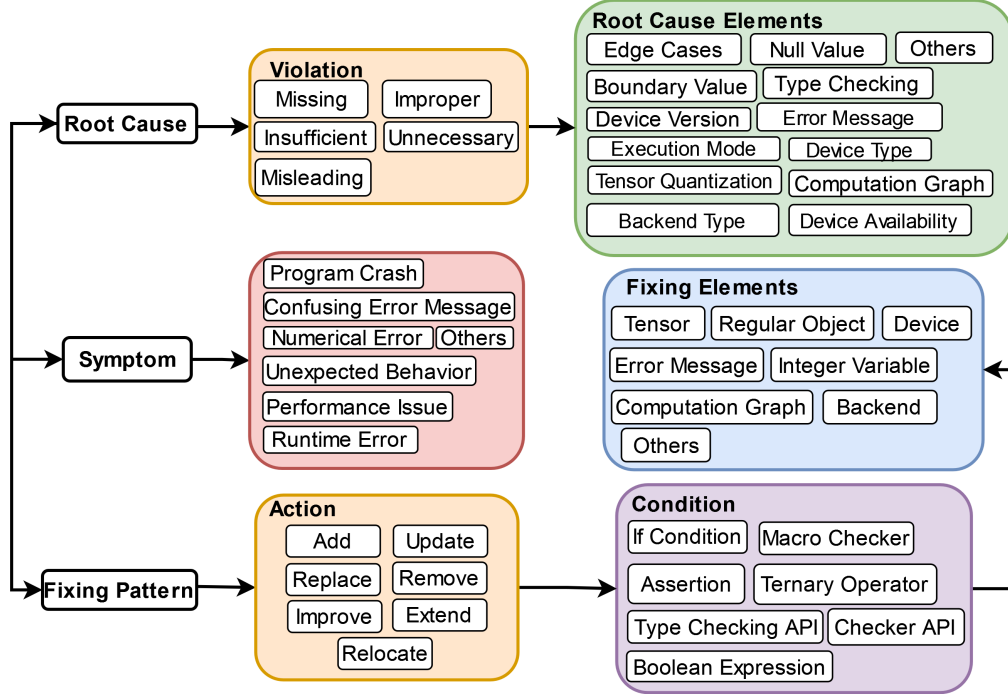


Figure 6.2: Taxonomy of DL checker bugs.

6.2.1 Study Setup

Selection of Subject DL Libraries: In this chapter, we selected TensorFlow [71] and PyTorch [187] as our subject libraries to characterize checker bugs due to their extensive usage in the literature [191, 192, 18, 193] and their widespread adoption in various application domains across academia and industry [81, 100, 194, 102, 103, 104, 105, 88, 106, 107]. PyTorch [187], originally developed by Meta, is an important machine-learning library based on the Torch library. TensorFlow [71], developed by Google Brain⁶², is crucial due to its robust and scalable framework that supports high-level and low-level tensor operations, making it suitable for a wide range of machine learning applications. Please note that we did not include other DL libraries, including Caffe [261], MLpack [74], Theano [262], and CNTK [263], as they have fewer checker-related bugs due to their lower popularity and maintainability. We did not include MXNet [73] because its GitHub repository is not maintained anymore.

⁶²<https://research.google.com/teams/brain/?authuser=2>

Incremental Keyword Construction: In this work, we used a snowball-based keyword-matching approach to collect checker bugs in DL libraries from commits in the main/master branches of the PyTorch and TensorFlow GitHub repositories. we started with 15 initial and general keywords associated with checker bugs used in previous work [49], e.g., *checker*, *validating*, *checking*, etc. We conducted a two-round snowball-based keyword construction with 1,000 and 10K different commits respectively. For each round we applied the existing keyword set to extract potential checker bugs, we then manually analyzed the commit messages and code changes to summarize possible new keywords. As a result, 40 and 47 new keywords were collected from each round.

Finally, we constructed a keyword set including 102 unique keywords. All keywords are available in the repository of this chapter⁶³.

Heuristic Commit Filtering: In this step, we use the final list of checker bug-related keywords to collect checker bugs from PyTorch and TensorFlow repositories at a large scale, i.e., from Sept. 2016 to Dec. 2023. At the end of this process, 1,045 related commits from PyTorch and 1,373 commits from TensorFlow were collected. Since heuristic filtering introduces false positives, we performed a manual analysis to filter irrelevant commits from our data, which is explained in detail in the next subsection.

Manual Filtering: The manual analysis was conducted collaboratively by three authors, who worked together to label each matched commit. Any disagreements during the analysis were resolved by simultaneous discussion and consensus. Each commit took approximately five minutes to label, involving the evaluation of commit messages, code changes, possible discussions within pull requests, and any other relevant contextual information. This approach resulted in 527 checker bugs, i.e., 221 from PyTorch and 306 from TensorFlow.

6.2.2 DL Checker Bugs Characteristics

We analyzed each checker bug from three major categories, i.e., root cause, symptom, and fixing pattern (shown in Figure 6.2). For our analysis, each checker bug was manually examined to identify its root cause, which involved understanding the underlying issue that led to the bug. We documented the symptoms by observing how the bugs manifested in the system, such as through error messages. Finally, we study the fixing patterns by reviewing the changes made in the commits that resolved these bugs and categorizing the different approaches to correct them.

⁶³<https://github.com/icsecs1992/TensorGuard>

Root Cause of DL Checker Bugs: We classify the root causes of checker bugs based on two sub-aspects, i.e., the root cause violation and root cause element. In the context of checker bugs, a violation type refers to the specific kind of bug that occurs in the code due to the absence or misconfiguration of certain checks or conditions. Root cause elements refer to the specific aspects or conditions within the code being verified or validated to ensure correct functionality, security, or performance [264, 57].

Root Cause Violation Types: Table 6.1 shows the distribution of the root causes of the DL checker bugs. In our taxonomy, there are five types of violations: *Missing*, *Improper*, *Insufficient*, *Unnecessary*, and *Misleading*. The *Missing* violation indicates the absence of a necessary root cause element. *Improper* involves incorrect use of root cause elements. *Insufficient* refers to inadequate or incomplete conditions that do not thoroughly validate the input, state, or conditions. *Misleading*, involves uninformative error messages that confuse users. Finally, *Unnecessary* indicates redundant root cause elements in the code base of DL libraries. Compared to Tian and Ray [49], our study presents three more new violation types, which are unique compared to traditional software.

Table 6.1: Distribution of root causes of DL checker bugs.

Root Cause Element	Root Cause Violation					Total
	Missing	Improper	Insufficient	Misleading	Unnecessary	
Edge Cases	164	46	23	-	4	237
Type Checking	37	16	9	-	2	64
Null Value	33	9	4	-	-	46
Boundary Value	22	8	3	-	-	33
Device Availability	17	3	8	-	1	29
Error Message	-	-	-	29	-	29
Device Type	7	4	3	-	2	16
Device Version	6	7	3	-	-	16
Execution Mode	7	3	-	-	2	12
Computation Graph	5	-	4	-	-	9
Tensor Quantization	5	1	1	-	1	8
Backend Type	5	1	1	-	-	7
Others	12	4	4	-	1	21
Total	320	102	63	29	13	527

Root Cause Element Types: We categorize root cause elements into 13 types. As shown in Table 6.1, *Edge Cases*, is the most frequent root cause element. The edge case refers to a scenario or input at the end of the operational parameters or beyond the typical scope of inputs that the implementation of DL APIs is designed to handle. *Type Checking* involves scenarios where the back-end implementation of

DL libraries has technical shortcomings in terms of checking or validating types of tensors or any other object. *Null Checker* refers to the improper handling of null values within DL libraries. The next root cause is *Boundary Check* which involves bugs related to ensuring that the values fall within the acceptable limits of input tensors, stacks, or arrays. These bugs occur when the program fails to correctly handle values at the edges of allowable input ranges. *Misleading Error Message* refers to scenarios where the error message of conditions or checks is not informative enough which confuses the DL library users. *Device Availability* refers to scenarios where the code fails to validate if the target device is available for tensor execution. *Device Version* typically refers to issues due to differences in device models, that is, GPU models, driver versions, or supported features. The next root cause element is *Device Type* refers to scenarios where the back-end fails to check which device type the tensor is running. *Tensor Execution Mode* refers to issues arising from the way tensor operations are executed in DL libraries including eager execution or graph execution. *Computation Graph* refers to issues that arise when the computational graph is not properly checked for correctness before execution. *Tensor Quantization* refers to scenarios in which the code does not validate if the tensors are executed in quantized mode. Lastly, *Backend Type* refers to scenarios where code fails to verify backend type, for example, Triton Backend⁶⁴. *Others* represents situations where the available data cannot identify a specific root cause element. When information is lacking, these cases are classified in the general category of *Others*.

Finding 1: Root cause violation types *Missing* and *Improper* are the most common violation types accounting for 60.7% and 19.3% instances in the experiment dataset, as for root cause elements, *Edge Cases* and *Type Checking* are the most two common root cause elements, accounting for 44.9% and 12.1% of instances in the experiment dataset.

Symptom of DL Checker Bugs: As shown in our taxonomy (see Fig. 6.2), the symptoms are classified into eight categories, i.e., *Program Crash*, *Unexpected Behavior*, *Confusing Error Message*, *Version Incompatibility*, *Numerical Error*, *Performance Issue*, *Runtime Error*, and the rest of instances as *Others*. Table 6.2 shows the detailed distribution of symptom categories per library. In the following paragraphs, we describe each symptom category in detail.

Program Crash: Is the most common symptom of checker bugs in DL libraries. A program crash is easy to identify because of the immediate sign of failure which

⁶⁴<https://github.com/triton-inference-server/backend>

is propagated in the log message or stack trace. Such signs of failure are included but not limited to: *Aborted (core dumped)*, *Assertion failed*, *Floating point exception (core dumped)*, *Segmentation fault*, and *Check failed*.

Unexpected Behavior: Refers to a program that does not function as expected. Checker bugs that include inappropriate handling of edge cases, such as empty tensors or missing data, or running DL APIs with the same inputs under different execution modes often result in unexpected behavior.

Confusing Error Message: Refers to misleading or unclear error messages introduced by DL checker bugs that can confuse users. This confusion arises because the bugs prevent the software from handling errors properly, making it difficult for users to understand or fix the checker bugs.

Performance Issue: This is a widespread symptom of checker bugs within DL libraries. These bugs often stem from issues related to tensor devices, i.e., either GPU or CPU, leading to slower execution times and increased resource consumption. They can manifest as prolonged training duration, excessive GPU/CPU utilization, or unexpected bottlenecks in data processing pipelines. Unlike other symptoms of checker bugs, performance issues do not generate a warning or log message. It often manifests itself in the form of slow execution of DL APIs and performance regression.

Numerical Error: Manifests itself in issues related to model accuracy or training stability. For example, operations on large tensors can result in unexpected integer overflows, potentially causing incorrect memory allocations or tensor manipulations. This is particularly problematic in complex neural architectures or when processing large datasets.

Others: Describes situations where the available data is inadequate to determine a particular symptom. When faced with insufficient information, we classify them under the broad category of *Others*.

Finding 2: *Program Crash* is the most common symptom of DL checker bugs accounting for 52.37% instances in total. *Unexpected Behavior* is the second most common symptom of DL checker bugs accounting for 28.46% instances.

Table 6.2: Distribution of symptom of DL checker bugs.

Symptom	TensorFlow	PyTorch	Total
Program Crash	199	77	276
Unexpected Behavior	62	88	150
Confusing Error Message	19	13	32
Performance Degradation	5	16	21
Numerical Error	9	2	11
Others	12	25	37
Total	306	221	527

Fixing Pattern: Following existing studies [55, 57], we consider a common fixing pattern consists of three components, i.e., *Action*, *Condition*, and *Elements*. *Action* refers to the specific modifications made to the code to resolve a checker bug. *Condition* refers to the type of condition used to fix a checker bug, e.g. if statements or macro-checkers. *Element* indicates the target component to which the fix is going to be applied.

Action Type: We categorize action types into seven categories (shown in Table 6.3): *Add*, which means the developer adds a condition to fix the checker bug; *Update*, indicating that an existing condition is modified; *Extend*, which refers to enhancing an insufficient condition to fix the bug; *Improve*, for example stating that the developer refines the error message to prevent confusing error message; *Replace*, which means the developer substitutes one condition type with another; *Relocate*, where the developer moves the condition within the codebase to fix the bug; and *Remove*, which involves the elimination of conditions to resolve the bugs. These categories encompass different approaches that developers take to fix checker bugs through condition modifications in the code.

Condition Type: We categorize the types of conditions (shown in Table 6.3) into seven categories: *If checker* indicates a typical if checker; *Macro Checker* refers to PyTorch specific or TensorFlow specific macro checkers, e.g., *OP_REQUIRE* for TensorFlow and *TORCH_CHECK* for PyTorch; *Type Checking API* refers to APIs that are used to match data types; *Assertion Statement*, which indicates a library specific assertion statement; *Checker API*, which indicates using an API to perform checking operation, e.g., *isCompatibleScope()*⁶⁵, *Ternary Operator* provides a concise way to perform a conditional check and return one of two values based on the result, which is represented by the ? and : symbols, and *Boolean Expression* is a logical

⁶⁵This checker API is responsible to make sure that nodes in computation graphs are compatible with each other.

statement that evaluates to either true or false, mostly performs checking on return values of functions.

Table 6.3: Distribution of conditions types.

Fixing Condition	Fixing Actions							Total
	Add	Extend	Update	Improve	Replace	Relocate	Remove	
If Checker	171	49	43	8	3	4	6	284
Macro Checker	126	6	20	14	11	2	7	186
Type Checking API	12	3	9	-	3	-	-	27
Assertion Statement	8	1	1	5	1	1	-	17
Checker API	2	1	2	-	1	-	-	6
Boolean Expression	-	3	1	-	-	-	-	4
Ternary Operatory	1	2	-	-	-	-	-	3
Total	320	65	76	27	19	7	13	527

Table 6.4: Distribution of fixing elements.

Fixing Elements	PyTorch	TensorFlow	Total
Tensor	73	184	257
Regular Object	38	47	85
Device	52	12	64
Error Message	13	16	29
Integer Variable	4	18	22
Computation Graph	7	2	9
Backend	6	1	7
Others	28	26	54
Total	221	306	527

Fixing Element: We categorize fixing elements into eight categories (shown in Table 6.4): *Tensor* refers to elements related to tensor properties, such as tensor shape (e.g., tensor element, tensor dimension, tensor rank, and tensor type), tensor storage, tensor quantization, execution mode, and boundary values on tensor properties. *Regular Object* includes general-purpose objects that are not tensors, such as custom classes or data structures. *Device* pertains to properties of devices, such as device storage, device version, and device availability. *Error Message* encompasses outputs generated when there is an issue with the execution in the backend of deep learning libraries. *Integer Variable* involves typical integer variables used in the backend implementation, whether as function or class arguments, or for indexing tensors, arrays, or buffers. *Computation Graph* represents the sequence of operations performed

in a deep learning model, ensuring the correctness of nodes and edges, including node type, value, or state. *Backend* refers to the type of backend that enhances the deployment of AI models in production, such as Triton or CUDA. *Others* includes elements that do not fit into the above categories.

Finding 3: *Add* and *Update* are the most common action types adopted to fix DL checker bugs accounting for 60.72% and 14.42% of instances, respectively. Among the condition types, *If Checker* and *Macro Checker* are the most common types accounting for 53.88% and 35.29% of instances, respectively. Among fixing elements, *Tensor* is the most common element accounting for 48.76% of instances.

6.3 TensorGuard: A RAG-based Multi-Agent Framework to Detect and Fix DL Checker Bugs

Our findings in Section 6.2 demonstrate that DL checker bugs primarily focus on issues related to tensor properties, device issues, and computation graphs. Examples include problems with edge cases, boundary values, device types, tensor types, and device availability. These issues typically fall outside the detection capabilities of current tools that target checker bugs in traditional software, as these tools generally rely on matching general checker templates rather than DL-specific ones [49, 50, 51, 52]. Identifying such checker bugs often requires manually crafted rules developed by library developers or experts in software security. However, manually crafted rules present scalability challenges.

Motivated by the notable progress made recently in LLMs [265] for software engineering tasks [266, 57, 267, 268], we develop **TensorGuard**, the first DL checker bug detector and fixing tool built on top of LLMs which is based on OpenAI GPT-3.5-turbo [269], stands out among similar LLMs in that it has shown great promise in software engineering [269].

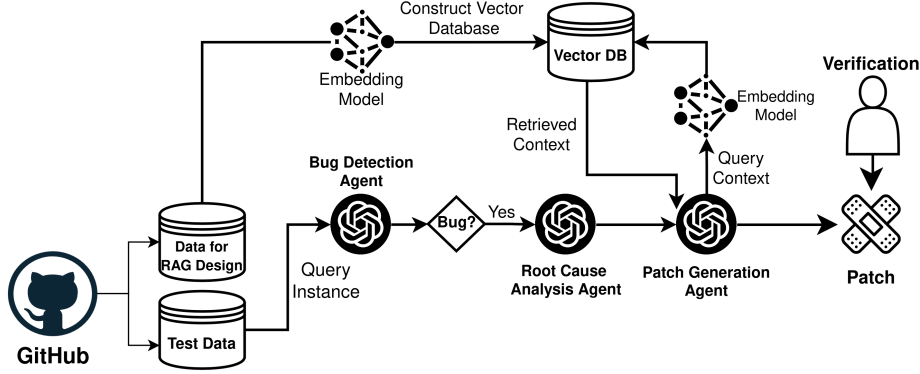


Figure 6.3: Architecture of TensorGuard.

6.3.1 Design of TensorGuard

In this section, we explain the design of **TensorGuard**. We start by detailing the design of the vector database for RAG. We then explain the prompt strategies and agents for checker bug detection and repair.

RAG Design: As shown in Fig. 6.3, **TensorGuard** is equipped with a vector database that is utilized in RAG that provides external knowledge for patch generation. Using RAG with our patch generation agent combines the strengths of retrieval-based methods and generative LLMs. The input to the RAG database is the root cause of the checker bug queried by the patch generation agent, and the output is a similar code change. By retrieving relevant contextual information from a large corpus of code changes, **TensorGuard** understands the context and specifics of the patch generation process, resulting in contextually relevant code fixes. To build the vector database, following existing work [270, 271], we used *sentence-transformers* library [272, 273] with *all-MiniLM-L6-v2* as the embedding model, which converts documents to a 384-dimensional dense vector space. The documents in our dataset are the whole body of code changes within each modified file, that is, including the deleted and added lines. To encode the code changes, we use *batch_size* as 50 and *chromadb*⁶⁶ as our database to store the vectorized code changes.

Prompt Engineering for Different Agents: In this section, we detail the prompts used in the agents for DL checker bug detection and repair tasks. **Checker Bug Detection Agent:** The purpose of developing this agent is to enable **TensorGuard** to identify potential checker bugs in a new commit. As shown in Fig. 6.3, the detection agent queries a code commit containing a commit message and code change. If

⁶⁶<https://pypi.org/project/chromadb/>

Table 6.5: Prompt template for bug detection agent (COT).

“prompt”: You are an AI trained to detect bugs in a deep-learning library based on commit messages and code changes. Your task is to determine whether a given commit introduces a bug or not. Follow the steps below to reason through the problem and arrive at a conclusion. 1. Understand the commit message: Analyze the commit message to understand the context and purpose of the code change. <code>{commit_message}</code> 2. Review the code change: Examine the deleted and added lines of code to identify the modifications made. <code>{code_removed}</code><code>{code_added}</code> 3. Identify potential issues: Look for any missing, improper, or insufficient checkers within the code change. Checkers may include error handling, input validation, boundary checks, or other safety mechanisms. 4. Analyze the impact: Consider the impact of the identified issues on the functionality and reliability of the deep learning libraries. 5. Make a decision: Based on the above analysis, decide whether the commit introduces a bug or not. 6. Output the conclusion: Generate a clear output of “YES” if the commit introduces a bug, or “NO” if it does not. “output”: <code>{Decision}</code>
--

the agent identifies the code changes contain bugs related to checkers, it passes the instance to subsequent agents, i.e., the root cause analysis agent. For this agent, we designed three prompting strategies including Chain of Thought (COT) [274], Zero-Shot [275], and Few-Shot[276] which are shown in Table 6.5, Table 6.6, and Table 6.7 respectively.

Our COT prompts were designed based on the knowledge gained from our empirical study on DL checker bugs, as illustrated in Fig. 6.2. Specifically, we develop the COT prompts through a step-by-step process to help the model understand the root cause of each bug. Then, we guided the agent to understand the impact of the checker bug. Note that for Few Shot, we randomly selected two examples from our curated dataset of 527 checker bugs in Section 6.2.

Root Cause Analysis Agent: The prompt template for this agent is shown in Table 6.8. This agent is responsible for extracting the root cause of the checker bug allowing the patch generation agent to create more targeted and effective fixes. The input to this agent is the commit message and the output is the root cause explanation of the DL checker bug.

Table 6.6: Prompt template for bug detection agent (Zero Shot).

“prompt”: You are an AI trained to detect bugs in a deep-learning library based on commit messages and code changes. Your task is to determine whether a given commit introduces a bug or not. Follow the steps below to reason through the problem and arrive at a conclusion.

Commit message: {commit_message}

Code change: {code_removed}{code_added}

“output”: {Decision}

Table 6.7: Prompt template for bug detection agent (Few Shot).

“prompt”: You are an AI trained to detect bugs in a deep-learning library based on commit messages and code changes. Your task is to determine whether a given commit introduces a bug or not. Follow the steps below to reason through the problem and arrive at a conclusion.

Example Checker Bug One:

Commit message: {commit_message}

Code change: {code_removed}{code_added}

Example Checker Bug Two:

Commit message: {commit_message}

Code change: {code_removed}{code_added}

Task:

Commit message: {commit_message}

Code change: {code_removed}{code_added}

“output”: {Decision}

Table 6.8: Prompt template for root cause analysis agent.

“prompt”: Please describe the root cause of the bug based on the following commit message:{commit_message}

“output”: {Root causes}

Patch Generation Agent: This agent is responsible for generating a fix for a DL checker bug. The prompt template is shown in Table 6.9. Its prompt is designed to

Table 6.9: Prompt template for patch generation agent.

“prompt”: You are given a bug explanation and an external context for fixing a checker bug. Please think step by step and generate a patch to fix the bug in the code snippet. Please neglect any issues related to the indentation in the code snippet. Fixing indentation is not the goal of this task. If you think the given pattern can be applied, generate the patch.

Example One: {code_removed} {code_added}

Example Two: {code_removed} {code_added}

Bug explanation: {bug_explanation}

Retrieved context: {retrieved_knowledge}

Code snippet: {code_snippet}

“output”: {Think steps}{Patch}

guide the agent in generating fixes for checker bugs. It provides the agent with a bug explanation (received from the root cause analysis agent), external knowledge (retrieved from the RAG database which plays a crucial role in enhancing the agent’s ability to generate accurate and contextually appropriate patches for checker bugs), and a code snippet containing the checker bug. The prompt instructs the model to think step-by-step and generate a patch. Additionally, the prompt includes two example shots to help the agent understand the task structure. The agent is expected to output its thinking steps followed by the generated patch.

Table 6.10: Statistics of data used for the experiments.

	Total Commits	Collection Interval	Number of Bugs	Clean Instances	Code changes
RAG Data	211K	2016/09 - 2023/12	-	-	1.3M
Evaluation Data	1,174	2024/01 - 2024/07	92	135	-

6.3.2 Data for RAG and TensorGuard Evaluation

Table 6.10 summarizes the data we used to build RAG’s database and evaluate the performance of TensorGuard.

RAG Data: To build the vector database for RAG, we collected the commits in PyTorch and TensorFlow GitHub repositories from January 2016 to December 2023. Note that, different from our empirical study conducted in Section 6.2 (which focused on checker-related commits), we used all the commits for building RAG, the reason is

that including all commits provides a more comprehensive dataset, allowing the RAG to learn from a wider variety of code changes and bug fixes. This broader context improves the accuracy and relevance of the patches generated, as it encompasses diverse scenarios and edge cases beyond just checker-related issues. This process yielded 61,453 commits for PyTorch and 150,352 commits for TensorFlow. After extracting commits, we started parsing each commit information and collected 391,571 code changes for PyTorch and 920,108 code changes for TensorFlow. Overall, we constructed the vector database with 1.3M code changes.

Test Dataset for TensorGuard: To build the evaluation data for **TensorGuard**, we applied our keyword-matching approach to the commits of PyTorch and TensorFlow from January 1, 2024 to July 20, 2024. We filtered commits that have a large number of co-changed files (i.e., more than 10 modified files) and changes that have large changes of modified code (i.e., changes with more than 15 lines of code). When provided with a large input, the model used, i.e., GPT-3.5-turbo may struggle to maintain focus on relevant parts of the code change, i.e., parts where checker conditions reside. It can be overwhelmed by the sheer volume of code-related information that is not related to checker bugs, making it difficult to identify and prioritize the most important checker bug-related details. As a result, the test data contains 92 buggy and 135 clean DL checker-related changes.

6.3.3 Experiments Setup

We assess **TensorGuard**’s effectiveness in detecting and repairing DL checker bugs. We use Precision, Recall, F1 score, and the number of correctly generated patches as the metrics to evaluate the performance. We chose to use GPT-3.5-turbo, as it has proven effective in software engineering tasks, particularly code generation[277].

We set the *temperature* parameter to 0 to ensure the determinism of LLM’s outputs [278]. Note that, due to the stochastic nature of LLMs, which can produce different outputs for the same input, we run **TensorGuard** five times on the test data and measure the average performance.

Baseline: To evaluate the effectiveness of **TensorGuard**, we compare its patch generation performance against *AutoCodeRover* [279], a prominent baseline from the SWEBench Leaderboard⁶⁷ at the time we conducted this research [280]. This decision was motivated by several factors that make *AutoCodeRover* particularly suitable for our study. First, its open-source nature allows transparent evaluation and reproduction of results. Secondly, *AutoCodeRover* is adaptable to repository-level data, such as issue reports and commit messages, making the comparison more

⁶⁷<https://www.swebench.com/>

straightforward, as **TensorGuard** also works on repository-level data. Secondly, it is standing as the second-best tool on the SWEBench, underscoring its competitive performance. Lastly, *AutoCodeRover*’s multi-agent architecture bears similarities to **TensorGuard**, providing a more direct comparison of approaches.

AutoCodeRover can be adjusted to work with various foundational models from different vendors, including OpenAI, Anthropic, Google, and open-source models from Meta. In this experiment, we opt to choose GPT-4o-mini from OpenAI because **TensorGuard** also operates with the GPT-3.5-turbo model. We use GPT-4o-mini because it has a 128K context window, which is suitable for *AutoCodeRover*, as its context information for patch generation is much larger compared to **TensorGuard**. We also set the temperature value to 0.2 according to their instructions. The input data to *AutoCodeRover* typically consists of GitHub issue descriptions, while our study focuses on commit messages. To adapt *AutoCodeRover* for our experiments, we did not directly feed the commit messages to it. Instead, we extracted the root cause of the bug from the commit messages and provided this information to *AutoCodeRover*. The rationale behind this approach is twofold: first, extracting the root cause from commit messages makes it similar to an issue description; second, by only extracting the root cause, we ensure that no hints for fix generation are included in the issue description.

Table 6.11: Performance of **TensorGuard** in detecting DL checker bugs using different prompting strategies.

	Chain of Thought			Zero-Shot			Few-Shot		
	Precision (%)	Recall (%)	F1 (%)	Precision (%)	Recall (%)	F1 (%)	Precision (%)	Recall (%)	F1 (%)
PyTorch	50.75	100	67.33	66.47	79.34	72.33	69.30	37.04	48.74
TensorFlow	30.05	89.03	44.93	57.46	63.22	60.20	55.60	50.96	53.17
Average	40.4	94.51	56.12	61.96	71.28	66.26	62.45	44.00	50.95

6.3.4 Experiments Results

Detection Effectiveness: Table 6.11 shows the performance **TensorGuard** in detecting checker bugs across PyTorch and TensorFlow averaged over five runs using three prompting strategies, i.e., COT, Zero-Shot, and Few-Shot.

Chain of Thought (COT) Prompting: With COT prompting, **TensorGuard** exhibits high recall (100%) but moderate precision (50.75%) on PyTorch’s data, leading to a relatively good F1 score of 67.33%. This indicates that while **TensorGuard** identifies nearly all bugs (high recall), it also produces many false positives (low precision). In TensorFlow, **TensorGuard** shows a high recall (89.03%) but very low precision (30.05%), resulting in an F1 score of 44.93%. The overall average precision

is 40.4%, the recall is 94.51%, and the F1 score is 56.12%. The high recall average implies that the tool **TensorGuard**, using Chain of Thought prompting, is a desirable tool for detecting checker bugs, as it identifies a large proportion of actual positive cases. This is due to the structured reasoning in COT prompting, which helps the checker bug detection agent break down the decision-making process into steps, potentially allowing it to consider more aspects of the problem and catch more potential issues [274].

Zero Shot Prompting: Under this setting, **TensorGuard** balances both metrics, with a precision of 66.47% and recall of 79.34%, leading to an F1 score of 72.33% in PyTorch. This indicates a good trade-off between identifying bugs and reducing false positives. In TensorFlow, **TensorGuard** shows improved balance as well, with precision at 57.46% and recall at 63.22%, yielding an F1 score of 60.20%. Precision averages at 61.96%, recall at 71.28%, and F1 score at 66.26%, indicating a well-rounded performance with a good balance between precision and recall.

Few Shot Prompting: Under this setting, **TensorGuard** achieves the highest precision among the three prompting strategies (69.30%) but suffers from low recall (37.04%), leading to an F1 score of 48.74% in the PyTorch library. In the TensorFlow, **TensorGuard** shows similar trends with a precision of 55.60% and recall of 50.96%, resulting in an F1 score of 53.17%. On average, **TensorGuard** reports a precision of 62.45%, a recall of 44.00%, and an F1 score of 50.95% indicating a preference for precision but with significant gaps in recall.

Detection Effectiveness: Overall, **TensorGuard**’s performance on detecting DL checker bugs varies significantly across strategies and frameworks. Chain of Thought excels in recall, Zero-Shot provides balanced performance, and Few-Shot favors precision.

Repair Effectiveness: Table 6.12 shows the performance of **TensorGuard** in patch generation. We manually verified the generated patch against the ground truth. Only patches that are semantically equivalent to the ground truth patches are considered valid. Note that, some of the generated patches included the same code elements as the ground truth but with additional comments. We considered these patches valid because the comments did not alter the logic of the fix. As shown in the table, **TensorGuard** generated a total of 90 patches, with 10 of them being correct patches, its accuracy is 11.1%. In contrast, *AutoCodeRover* generated 32 patches while only three were correct and its accuracy is 9.3%. This suggests that **TensorGuard** can outperform *AutoCodeRover* in patching DL checker bugs.

Note that, *AutoCodeRover* generates fewer patches compared to **TensorGuard**, this is due to an issue regarding the size of the context information *AutoCodeRover* retrieves from the target project’s repository. *AutoCodeRover* parses hundreds of source files within the target project to gather contextual information for patch generation [279]. Although we used GPT-4o-mini as its base model (with 128K context window size), for some large commits, the context length exceeds the model’s window size. This results in halting the execution of *AutoCodeRover* without generating any patch.

In addition, regarding the accuracy of generated patches, *AutoCodeRover* has a lower accuracy (i.e., 9.3%) compared to its accuracy on general GitHub issues from the SWEBench dataset (i.e., 18.83%). This discrepancy can be attributed to several factors. Firstly, the SWEBench dataset includes general bugs that might be easier to fix due to their more common and straightforward nature. The diverse range of bugs in SWEBench provides a broader spectrum of examples that the model can leverage, making it more likely to encounter familiar patterns and generate accurate fixes. Secondly, we manually checked that most issues in the SWEBench dataset are from before 2023, which means that these bugs have probably been seen by LLMs during the training.

The following code change shows that **TensorGuard** successfully generated a correct fix for a DL checker bug. The root cause of this bug was the failure to check if the type of the tensor element was an integer (lines 2 and 3). **TensorGuard** resolved this bug by adding an if-statement to ensure that the tensor element is of an integer type.

```

89 # Checker bug code snippet
90 + return element_type.isInteger() &&
91 +     (element_bitwidth == 32 ||
92 element_bitwidth == 64);
93
94 # Actual patch made by the developer
95 + if (!element_type.isInteger()) {
96 +     return false;}
97 + return element_bitwidth == 32 ||
98 element_bitwidth == 64;
99
100 # Correct patch generated by TensorGuard:
101 + if (!element_type.isInteger()) {
102 +     return false;}
103 + return element_bitwidth == 32 ||

```

```
104 element_bitwidth == 64;
```

We also show a typical incorrect patch generated by **TensorGuard** as follows. The root cause of the bug was an overflow issue in the comparison of *bytes_to_write* and *AvailableInputSpace()*. The variable *bytes_to_write* could overflow int32, making it appear smaller than the actual *AvailableInputSpace()*. **TensorGuard** failed to generate a correct patch because it incorrectly cast *bytes_to_write* instead of casting *AvailableInputSpace()*.

From the example, we can see that **TensorGuard** could be further improved by providing a detailed root cause analysis before generating patches, and also incorporating a user feedback loop can help continuously improve the effectiveness of the tool in code generation. User feedback can be retrieved from code documentation like discussions among developers or issue descriptions, which can help **TensorGuard** understand the broader context and specifics of the checker bugs that may not be evident from the commit message and code changes.

```
89 # Checker bug code snippet
90 + if (static_cast<int32>(bytes_to_write) <=
91 AvailableInputSpace()) {
92
93 # Actual patch made by the developer
94 + if (bytes_to_write <=static_cast<size_t>
95 (AvailableInputSpace())) {
96
97 # Incorrect patch generated by TensorGuard
98 + if (static_cast<size_t>(bytes_to_write)
99 <= AvailableInputSpace()) {
```

Repair Effectiveness: Despite the limited effectiveness in fixing DL checker bugs, **TensorGuard** can outperform state-of-the-art bug repair tool, i.e., *AutoCodeRover* in patching DL checker bugs.

Table 6.12: Performance of **TensorGuard** versus *AutoCodeRover* in fixing DL checker bugs.

Approach	# Generated Patch	# Correct Patches	Accuracy
<i>AutoCodeRover</i>	32	3	9.3%
TensorGuard	90	10	11.1%

6.4 Discussion

6.4.1 Detecting New Checker Bugs

We also assess the effectiveness of **TensorGuard** in finding new checker bugs. Specifically, we experiment with a well-known and widely used DL library, i.e., JAX from Google⁶⁸. We first collected the checker bug-related commits from January 2024 to July 2024 based on our keyword-matching approach, which resulted in 87 commits encompassing 493 code changes. We then ran **TensorGuard** on these code changes. Once the execution was completed, we manually analyzed the results in two stages for further verification. In the first stage, we checked whether each detected code change was related to checker bugs. If a code change was verified as a true positive, we proceeded to the next stage, in which we assessed whether the generated patch was correct.

As shown in Table 6.13, out of a total of 493 changes, **TensorGuard** labeled 118 instances as potential checker bugs. Upon further manual inspection, 64 of them were verified to be true DL checker bugs. However, **TensorGuard** successfully generated correct fixes for only four of these true checker bugs. This indicates that while the tool is relatively effective at identifying potential checker bugs, its success rate in generating accurate fixes for these bugs is significantly lower. The relatively low success rate of **TensorGuard** in generating accurate fixes for the detected checker bugs in the JAX library can be attributed to the limitations of the RAG database used in **TensorGuard**. The current RAG database is built primarily on data collected from TensorFlow and PyTorch repositories, which might not encompass a wide range of fix patterns applicable to checker bugs within the JAX library. These repositories, while extensive, may contain specific code practices and patterns that do not generalize well in the JAX library.

Table 6.13: Performance of **TensorGuard** on detecting and fixing new checker bugs in JAX library.

	Total changes	# Detected	# True Bugs	# Correct Fixes
JAX	493	118	64	4

⁶⁸<https://github.com/google/jax>

6.4.2 Difference Between Checker Bugs in DL Libraries and Traditional Software

Differences in Root Causes. The root causes of DL checker bugs differ significantly from those of traditional software, both in terms of root cause violations and root cause elements. For example, Tian and Ray [49] categorized checker bug violations into two types: *Incorrect* and *Missing*. Although these two categories are frequent among DL checker bugs, we found three new violation types during our manual analysis: *Insufficient*, *Misleading*, and *Unnecessary* resulting in five categories in total. In terms of root cause elements, Tian and Ray [49] provided four categories for root cause elements, i.e., *Error Checks (EC)*, *Error Propagation (EP)*, *Error Output (EO)*, and *Resource Release (RR)*. In our taxonomy, we identify a total of 13 categories. In traditional software, the root cause of checker bugs is typically related to common programming defects. However, within DL libraries, as shown in Fig. 6.2, the root cause elements are more comprehensive and distinct. For example, *Edge Case* primarily refers to malicious values within tensor properties, such as tensor shape, value, or data type. Additionally, DL libraries have unique root cause elements like *Computation Graph* and *Tensor Quantization* [53].

Differences in Symptoms: Symptoms of checker bugs are also different between traditional software and DL libraries. To show the difference, we compared it with the study conducted by Lu et al. [53]. They categorized the security symptom of missing-check bugs into six categories including *DoS*, *Overflow*⁶⁹, *Bypass Privilege*, *Information Leak*, *Memory Corruption*, and *Code Execution*. Compared to their study, we have five distinct symptoms within DL libraries including *User Confusion*, *Numerical Error*, *Version Incompatibility*, *Runtime Error*, and *Unexpected Behavior*.

Differences in Fixing Patterns: Tian and Ray [49] proposed *ErrDoc* including four distinct fixing patterns to fix checker bugs in C programs based on AST editing, which generally follows a common procedure, i.e., given a pattern and a matched bug, *ErrDoc* first identifies the bug location, then it adds necessary checks and error handling blocks, substitutes error-specific constants, and inserts appropriate logging or paired function calls. Compared to the patterns discovered by Tian and Ray [49] in C programs, the fixing patterns for checker bugs in DL libraries are dramatically different. Specifically, the fixing patterns for DL checker bugs primarily involve ensuring that the shape of input tensors is valid, meaning input tensors must have the same shape or align correctly with the corresponding index parameters within DL APIs. Existing work [57] showed that DL APIs typically have more parameters than traditional APIs due to the sophisticated tasks they handle, such as convolutional

⁶⁹*Overflow* category in their study refers to buffer overflow.

operations. The increased number of parameters, along with the complex correlations among them [281, 47, 48], introduces many critical weaknesses within the DL back-end implementations. Therefore, developers must guard against these conditions to enhance the reliability of DL systems.

6.4.3 Guidelines to Avoid DL Checker Bugs

Ensure Input Tensors have the Same Shape: Based on our findings, we suggest that the developers of TensorFlow and PyTorch add the necessary checks to ensure that the input tensor shapes are aligned with each other according to the constraints within the API documentation. The location of the checkers is really important. Based on our analysis we suggest developers insert checkers at the beginning of the functions because most checks should be done before main computations are started.

Check for Tensor Types: Based on our analysis, we suggest developers add type-checking conditions to ensure tensor types are valid, either one single tensor or a mismatch between types of input tensors. For example, in some scenarios, multiple input tensors need to be numeric, i.e., the type of input tensors should match. Additionally, there are cases where single tensors must follow specific types for further computations.

Ensure Tensor Indexing is Within Range: Based on our findings, we suggest that TensorFlow backend developers must prioritize ensuring tensor indexing remains within valid ranges to prevent out-of-bounds errors. Implement rigorous checks for index validity, especially when handling sparse tensors or complex indexing operations. Utilize built-in TensorFlow functions for boundary enforcement where possible and incorporate explicit range validation in custom operations. When processing user-provided indices, always verify their consistency and order. For instance, validate that indices are sorted in ascending order for sparse tensor operations. Additionally, it employs defensive programming techniques such as bounds-checking loops, error handling for potential out-of-range access, and clear error messages for invalid input.

6.5 Threats to Validity

In our empirical study, we focus on checker bugs stemming from the backend implementation of DL libraries which often are developed in Python, C/C++, and CUDA languages. We exclude DL applications developed in front-end languages like Python and Java Script as the majority of checker bugs occur within the backend implementation of DL libraries. To ensure the internal validity of our manual analysis,

the first three authors with more than five years of DL application development conducted the analysis together in multiple rounds. This collaborative approach allowed for immediate discussion and resolution of any disagreements or discrepancies. By working simultaneously, the authors were able to cross-verify each other’s assessments in real-time, ensuring consistency and accuracy in our findings. One major threat to the external validity of this chapter is our exclusive focus on TensorFlow and PyTorch. We believe our findings apply to other DL libraries like Caffe, Theano, and MXNet, due to their common use of tensor data structures, computation graphs, and high-performance computing devices like GPUs. To ensure we had the most up-to-date checker bug patterns for evaluating **TensorGuard**, we collected commits from the last seven months. For RAG construction and taxonomy creation, we selected commits over eight years to ensure comprehensive coverage of checker bug patterns, rather than limiting our analysis to a specific timeframe. In this work, we experimented with ChatGPT-3.5-turbo. It is essential to recognize other LLM models, such as Meta’s Llama2, LLaMa 3.1, Mistral, Google PaLM/Gemini, etc. These alternative models may bring unique features and performance characteristics that could potentially impact the validity of our findings.

6.6 Conclusion

In this chapter, we conducted the first comprehensive analysis of DL checker bugs in two major DL libraries, TensorFlow and PyTorch. We curated a data set including 527 checker bugs through manual inspection. Our analysis of these DL bugs, focusing on their root causes, symptoms, and fixing patterns, has provided valuable insights.

Leveraging these findings, we created **TensorGuard**, a proof-of-concept tool powered by RAG and LLM agents to identify and fix checker bugs in DL libraries. Our evaluation of a test dataset that includes 92 buggy and 135 clean checker-related changes in TensorFlow and PyTorch from January 2024 to July 2024 demonstrates **TensorGuard**’s effectiveness. Furthermore, **TensorGuard** helps detect 64 new DL checker bugs for the JAX library from Google. In general, our results underscore the potential of **TensorGuard** to improve the reliability of DL libraries by effectively identifying and fixing checker bugs. This work sets the stage for future advances in bug detection and correction for DL systems, contributing to more robust and efficient AI applications.

6.6.1 Data Availability

All the scripts and data to reproduce our experiments can be found in our replication package⁷⁰.

⁷⁰<https://github.com/icsecs1992/TensorGuard>

Chapter 7

Conclusions and Future work

Although AI infrastructure plays a key role in our daily lives, ensuring its reliability and trustworthiness remains a significant challenge. This thesis takes the first step toward addressing these challenges and enhancing the reliability of AI infrastructure through a systematic analysis of software repositories. We derive guidelines for developing high-quality AI system code and empirically evaluate various approaches related to AI reliability. The findings and techniques presented in this work benefit both AI framework developers and end users.

This chapter is organized as follows: first, we summarize our findings and contributions based on the presented work in this thesis (Section 7.1). Then, we discuss some future works (Section 7.2). At last, we provide our closing remarks (Section 7.3).

7.1 Thesis Findings and Contributions

- **Characterizing Bugs in AI Frameworks:** We proposed the first empirical study to characterize software bugs in AI frameworks, aiming to guide the design and development of effective bug detection and debugging techniques. By manually analyzing 683 commits from seven widely-used AI frameworks—TensorFlow, PyTorch, Scikit-learn, Mlpack, Scipy, Pandas, and Numpy—we identified 16 bug types, 20 root causes, 4 symptoms, and 19 fixing patterns, culminating in six key findings. Based on these insights, we propose actionable guidelines to enhance the reliability and security of AI frameworks, providing valuable contributions to the ML and software engineering communities.

- **Static Bug Detection in AI Frameworks:** We investigated the automatic detection of software security bugs in AI frameworks by evaluating the effectiveness of five popular static analysis tools: Flawfinder, RATS, Cppcheck, Facebook Infer, and Clang Static Analyzer. Using a curated dataset of bugs from four widely used AI frameworks, we categorize these tools’ strengths and limitations.
- **Benchmarking Deep Learning Fuzzers:** We present the first evaluation of state-of-the-art DL fuzzers against real-world bugs in DL libraries. We curated a comprehensive benchmark dataset of 517 real-world bugs from PyTorch and TensorFlow that are detectable through fuzzing. Applying seven leading DL fuzzers—FreeFuzz, DeepRel, NablaFuzz, DocTer, ACETest, TitanFuzz, and FuzzGPT—we found that these tools collectively detected only 6.5% (34/517) of the bugs. Our analysis identified two key factors limiting their effectiveness, offering actionable insights for enhancing fuzzer design. We established a benchmark for DL fuzzing research and provided a foundation for developing more robust and efficient fuzzing tools.
- **History-Driven Fuzzing for Deep Learning Libraries:** We introduced Orion, a semi-automated fuzzing framework that leverages fuzzing heuristic rules derived from the historical security bugs of TensorFlow and PyTorch. By mining open-source data, including API documentation, GitHub repositories, and library tests, Orion generates test inputs based on 33 unique root causes of bugs. Evaluated over 5,000 DL APIs, Orion identified 135 bugs, with 76 confirmed by the DL community—69 of which are novel, and 7 have been fixed post-reporting. Compared to traditional fuzzers like DeepRel and state-of-the-art LLM-based fuzzers like AtlasFuzz, Orion achieves up to 90% and 18.42% higher detection rates, respectively. These results establish Orion as a significant advancement in DL library fuzzing research, paving the way for improved security and reliability.
- **Taming Checker Bugs in Deep Learning Libraries:** We conducted the first comprehensive analysis of DL checker bugs in TensorFlow and PyTorch, supported by a curated dataset of 527 checker bugs. We then proposed TensorGuard, a proof-of-concept tool combining RAG and LLM agents to identify and fix checker bugs in DL libraries. Evaluated on a dataset of 92 defective and 135 clean checker-related changes from January to July 2024, TensorGuard demonstrates high effectiveness and further identifies 64 new checker bugs in

Google’s JAX library. These results highlight TensorGuard’s potential to enhance DL library reliability by automating bug detection and correction, paving the way for more robust AI systems.

7.2 Future Work

- **Direction 1: Effectiveness of Memory Related Dynamic Analysis Tools on AI Frameworks:** In my Ph.D. studies, we applied static analysis tools to the implementation of AI frameworks to evaluate their effectiveness in detecting security bugs. Building on this, we plan to apply dynamic analysis tools, such as Valgrind [282], LLVM Address Sanitizer [283], and Google Address Sanitizer [284], to assess their ability to detect security bugs. The advantage of dynamic analysis tools [285] is that they monitor the system during execution, capturing actual runtime behavior, including memory allocation, access, and deallocation. This is particularly useful for detecting memory-related issues like buffer overflows, use-after-free errors, and memory leaks, which often depend on specific runtime conditions such as inputs, timing, or memory access patterns. In this direction, we want to explore what is the effectiveness of dynamic code analysis tools in finding security bugs within AI frameworks.
- **Direction 2: System-Level Fuzzing of AI Frameworks:** In this direction, we plan to develop system-level fuzzers for AI frameworks. In the context of fuzzing APIs in the realm of AI, simply invoking API functions with malicious inputs is often insufficient for comprehensive bug detection. The fuzzer must also be capable of modeling specific code patterns unique to AI code patterns, such as tensor operations, model architecture definition, or the use of external libraries (e.g., importing pre-trained models). These code patterns are integral to AI workflows and introduce unique challenges that require specialized fuzzing techniques. Recent LLM-based DL fuzzers, such as FuzzGPT [47] and TitanFuzz [48], are designed to support more diverse code patterns to enhance bug detection. However, our experiments reveal that they struggle to model complex code patterns in the test cases they generate. For instance, certain bugs can only be triggered when the test case imports external libraries and incorporates their APIs within the test case body, a complexity these fuzzers fail to address effectively.
- **Direction 3: Feedback-Driven Fuzzing of AI frameworks:** In my Ph.D. studies, we find that LLM-based fuzzers generate a substantial number of

failed or syntactically incorrect test cases. Although these fuzzers demonstrate superior bug detection capabilities, the high rate of failed or incorrect test cases raises concerns about their reliability in generating effective test cases. In this new direction, our objective is to investigate whether we can mitigate the failure rate of LLM-powered fuzzers by incorporating feedback from log messages generated by the test cases. The log messages associated with the generated test cases contain informative and meaningful insights that can be leveraged to improve the performance of the DL fuzzers. Through this exploration, our goal was to refine the reliability and effectiveness of LLM-powered DL fuzzers by integrating valuable feedback from log messages or stack traces.

- Direction 4: Assertion Generation for AI Framework Test Cases:** Assertion statements within test cases are fundamental for guaranteeing the accuracy and dependability of software [286]. This process facilitates the validation of whether the observed behavior aligns with the expected behavior. Additionally, assertion statements operate as validators for errors and disparities throughout testing. If an assertion fails during the execution of a test case, it signals the presence of an unforeseen condition, streamlining the identification and localization of issues in the code. Even though assertion generation for traditional software has been done in the literature [287], there is still a gap in assertion generation for AI test cases. In this direction, *We want to understand What is the effectiveness of LLMs in assertion generation within AI framework's test cases.*
- Direction 5: LLM-based Automatic Program Repair:** In my Ph.D. studies, we developed an LLM-based automatic program repair equipped with RAG architecture called TensorGuard to detect and fix checker bugs within AI frameworks. In this direction, our objective is to expand the scope and application of TensorGuard. First, we will enhance TensorGuard, by incorporating more advanced prompt engineering techniques and expanding its capabilities to cover a broader range of frameworks beyond TensorFlow, PyTorch, and JAX. This includes investigating how checker bugs manifest in emerging AI frameworks and extending the tool's detection and repair capabilities across different versions and configurations. we will also explore the integration of automated static and dynamic analysis techniques with LLM-based approaches to improve the precision and accuracy of bug detection and repair. Additionally, we plan to collaborate with the developer community to implement continuous monitoring systems to detect checker bugs in real time during the library development process. Furthermore, we plan to investigate the long-term impact of

undetected checker bugs on model performance, reliability, and security in AI systems.

- **Direction 6: Cross Platform Consistency Testing of AI Compilers:** AI compilers play a critical role in the efficient execution of AI models across diverse hardware platforms [76, 77, 78]. They optimize performance models, translating high-level representations into low-level code that can run on CPUs, GPUs [79], or specialized hardware like TPUs [76]. This is crucial because well-optimized AI compilers can dramatically enhance the speed and efficiency of AI applications, particularly in environments where performance is key, such as autonomous vehicles [80]. However, the diversity of hardware and compilers introduces complexity, and inconsistencies may arise when the same AI model is compiled and run on different platforms. This is where cross-platform consistency testing becomes essential. By testing the same model across multiple compilers (e.g., TensorFlow XLA [71], TVM [78], or ONNX [77]), we can ensure that the behavior remains consistent and correct, regardless of the target hardware. Discrepancies in outputs due to subtle variations in optimization, floating-point arithmetic, or operator implementations can introduce bugs or inaccuracies in critical applications.
- **Direction 7: Performance Benchmarking of AI Compilers:** Performance benchmarking of AI compilers is essential for understanding and maximizing the efficiency of AI models across various hardware platforms [79, 288]. Given the growing complexity of AI workloads and the diversity of available hardware, well-optimized AI compilers are vital to improving performance, reducing latency, and minimizing resource consumption. In this direction, we plan to perform performance benchmarking, which is critical in evaluating and comparing how different compilers handle the same model across various platforms. It helps identify which compilers are best suited for particular tasks or hardware environments, highlighting areas for improvement in memory usage, and energy efficiency. Without consistent performance benchmarking, developers and researchers may struggle to determine the most effective toolchain for deploying their models, potentially leading to suboptimal performance.

7.3 Closing Remarks

AI frameworks play a critical role in modern AI development, yet they are prone to unique software security bugs. This thesis explores automated approaches to improve

the reliability and security of AI frameworks. First, we characterize software bugs in AI frameworks, revealing 16 bug types, 20 root causes, and actionable guidelines for developers. We also assess the effectiveness of static analysis tools, uncovering their limitations in detecting bugs in AI frameworks. To address these gaps, we benchmark DL fuzzers, finding significant challenges in their detection capabilities and proposing improvements. Additionally, we introduce Orion, a history-driven fuzzing framework that outperforms state-of-the-art tools by leveraging insights from past bugs. Finally, we develop TensorGuard, a novel tool to identify and fix checker bugs using LLM and RAG techniques, demonstrating its impact through new bug discoveries in TensorFlow, PyTorch, and JAX. This thesis provides critical insights and methodologies to advance AI framework security and offers practical tools for developers and researchers.

Bibliography

- [1] Y.-Y. Chang, P. Zavarisky, R. Ruhl, and D. Lindskog, “Trend analysis of the cve for software vulnerability management,” in *2011 IEEE third international conference on privacy, security, risk and trust and 2011 IEEE third international conference on social computing*. IEEE, 2011, pp. 1290–1293.
- [2] P. Foreman, *Vulnerability management*. CRC Press, 2019.
- [3] A. Raza and W. Ahmed, “Threat and vulnerability management life cycle in operating systems. a systematic review,” *Journal of Multidisciplinary Engineering Science and Technology (JMEST)*, vol. 9, no. 1, 2022.
- [4] M. Walkowski, J. Oko, and S. Sujecki, “Vulnerability management models using a common vulnerability scoring system,” *Applied Sciences*, vol. 11, no. 18, p. 8735, 2021.
- [5] C. Gao, G. Wang, W. Shi, Z. Wang, and Y. Chen, “Autonomous driving security: State of the art and challenges,” *IEEE Internet of Things Journal*, vol. 9, no. 10, pp. 7572–7595, 2021.
- [6] Q. Luo, Y. Cao, J. Liu, and A. Benslimane, “Localization and navigation in autonomous driving: Threats and countermeasures,” *IEEE Wireless Communications*, vol. 26, no. 4, pp. 38–45, 2019.
- [7] K. Goseva-Popstojanova and J. Tyo, “Experience report: Security vulnerability profiles of mission critical software: Empirical analysis of security related bug reports,” in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 152–163.
- [8] N. S. Harzevili, J. Shin, J. Wang, and S. Wang, “Characterizing and understanding software security vulnerabilities in machine learning libraries,” *arXiv preprint arXiv:2203.06502*, 2022.

- [9] N. Lu, B. Wang, Y. Zhang, W. Shi, and C. Esposito, “Neucheck: A more practical ethereum smart contract security analysis tool,” *Software: Practice and Experience*, vol. 51, no. 10, pp. 2065–2084, 2021.
- [10] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, “Smart contract vulnerability analysis and security audit,” *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.
- [11] T. H. M. Le, B. Sabir, and M. A. Babar, “Automated software vulnerability assessment with concept drift,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 371–382.
- [12] O. Suciu, C. Nelson, Z. Lyu, T. Bao, and T. Dumitras, “Expected exploitability: Predicting the development of functional vulnerability exploits,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 377–394.
- [13] J. Jacobs, S. Romanosky, O. Suciuc, B. Edwards, and A. Sarabi, “Enhancing vulnerability prioritization: Data-driven exploit predictions with community-driven insights,” *arXiv preprint arXiv:2302.14172*, 2023.
- [14] J. Yin, M. Tang, J. Cao, H. Wang, and M. You, “A real-time dynamic concept adaptive learning algorithm for exploitability prediction,” *Neurocomputing*, vol. 472, pp. 252–265, 2022.
- [15] T. H. Le, H. Chen, and M. A. Babar, “A survey on data-driven software vulnerability assessment and prioritization,” *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–39, 2022.
- [16] V. Mahor, K. Pachlasiya, B. Garg, M. Chouhan, S. Telang, and R. Rawat, “Mobile operating system (android) vulnerability analysis using machine learning,” in *Proceedings of International Conference on Network Security and Blockchain Technology: ICNSBT 2021*. Springer, 2022, pp. 159–169.
- [17] S. Frei, M. May, U. Fiedler, and B. Plattner, “Large-scale vulnerability analysis,” in *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, 2006, pp. 131–138.
- [18] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, “Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks,” *arXiv preprint arXiv:2203.02660*, 2022.

- [19] S. Jeon and H. K. Kim, “Autovas: An automated vulnerability analysis system with a deep learning approach,” *Computers & Security*, vol. 106, p. 102308, 2021.
- [20] J. Tian, W. Xing, and Z. Li, “Bvdetector: A program slice-based binary code vulnerability intelligent detection system,” *Information and Software Technology*, vol. 123, p. 106289, 2020.
- [21] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang, “Cd-vuld: Cross-domain vulnerability discovery based on deep domain adaptation,” *IEEE Transactions on Dependable and Secure Computing*, 2020.
- [22] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, “Deepwukong: Statically detecting software vulnerabilities using deep graph neural network,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.
- [23] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “muvuldeepecker: A deep learning-based system for multiclass vulnerability detection,” *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.
- [24] N. Ziems and S. Wu, “Security vulnerability detection using deep learning natural language processing,” in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2021, pp. 1–6.
- [25] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, “Software vulnerability discovery via learning multi-domain knowledge bases,” *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2469–2485, 2019.
- [26] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “Sysevr: A framework for using deep learning to detect software vulnerabilities,” *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [27] V. Piantadosi, S. Scalabrino, and R. Oliveto, “Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat,” in *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 68–78.

- [28] Z. Chen, S. Kommrusch, and M. Monperrus, “Neural transfer learning for repairing security vulnerabilities in c code,” *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.
- [29] G. Bhandari, A. Naseer, and L. Moonen, “Cvefixes: automated collection of vulnerabilities and their fixes from open-source software,” in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39.
- [30] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.
- [31] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu *et al.*, “To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools,” in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 50–59.
- [32] —, “To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools,” *Automated Software Engineering*, vol. 22, no. 4, pp. 561–602, 2015.
- [33] A. Habib and M. Pradel, “How many of all bugs do we find? a study of static bug detectors,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 317–328.
- [34] D. A. Tomassi and C. Rubio-González, “On the real-world effectiveness of static bug detectors at finding null pointer exceptions,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 292–303.
- [35] S. Lipp, S. Banescu, and A. Pretschner, “An empirical study on the effectiveness of static c code analyzers for vulnerability detection,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 544–555.
- [36] V. J. Manès, S. Kim, and S. K. Cha, “Ankou: Guiding grey-box fuzzing towards combinatorial difference,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1024–1036.

- [37] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, “Memlock: Memory usage guided fuzzing,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 765–777.
- [38] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.
- [39] Facebook. (2019) American fuzzy lop. [Online]. Available: <https://github.com/google/AFL/releases>
- [40] K. Serebryany. (2015) libfuzzer a library for coverage-guided fuzz testing. [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>
- [41] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, “Mopt: Optimized mutation scheduling for fuzzers.” in *USENIX Security Symposium*, 2019, pp. 1949–1966.
- [42] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah, “Ems: History-driven mutation for coverage-based fuzzing,” in *29rd Annual Network and Distributed System Security Symposium, NDSS*, 2022, pp. 24–28.
- [43] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.” in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.
- [44] A. Wei, Y. Deng, C. Yang, and L. Zhang, “Free lunch for testing: Fuzzing deep-learning libraries from open source,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 995–1007.
- [45] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, “Docter: documentation-guided fuzzing for testing deep learning api functions,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 176–188.
- [46] Y. Deng, C. Yang, A. Wei, and L. Zhang, “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*, 2022, pp. 44–56.

- [47] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 423–435.
- [48] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, “Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt,” *arXiv preprint arXiv:2304.02014*, 2023.
- [49] Y. Tian and B. Ray, “Automatically diagnosing and repairing error handling bugs in c,” in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 752–762.
- [50] A. Pakki and K. Lu, “Exaggerated error handling hurts! an in-depth study and context-aware detection,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1203–1218.
- [51] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, “Understanding and detecting disordered error handling with precise function pairing,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2041–2058.
- [52] D. Zhan, X. Yu, H. Zhang, and L. Ye, “Errhunter: Detecting error-handling bugs in the linux kernel through systematic static analysis,” *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 684–698, 2022.
- [53] K. Lu, A. Pakki, and Q. Wu, “Detecting {Missing-Check} bugs via semantic- and {Context-Aware} criticalness and constraints inferences,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.
- [54] Y.-J. Wang, L.-Z. Yin, and W. Dong, “Amchex: Accurate analysis of missing-check bugs for linux kernel,” *Journal of Computer Science and Technology*, vol. 36, pp. 1325–1341, 2021.
- [55] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, “Characterizing and understanding software security vulnerabilities in machine learning libraries,” in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 27–38.
- [56] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

- [57] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, “Demystifying and detecting misuses of deep learning apis,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
- [58] S. Willcock, J. Martínez-López, D. A. Hooftman, K. J. Bagstad, S. Balbi, A. Marzo, C. Prato, S. Sciandrello, G. Signorello, B. Voigt *et al.*, “Machine learning for ecosystem services,” *Ecosystem services*, vol. 33, pp. 165–174, 2018.
- [59] J.-W. Hong, Y. Wang, and P. Lanz, “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*, vol. 36, no. 18, pp. 1768–1774, 2020.
- [60] Z. Lv, R. Lou, and A. K. Singh, “Ai empowered communication systems for intelligent transportation systems,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4579–4587, 2020.
- [61] L. Cao, “Ai in finance: challenges, techniques, and opportunities,” *ACM Computing Surveys (CSUR)*, vol. 55, no. 3, pp. 1–38, 2022.
- [62] J. Wang, T. Sun, B. Liu, Y. Cao, and D. Wang, “Financial markets prediction with deep learning,” in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 97–104.
- [63] M. Hiransha, E. A. Gopalakrishnan, V. K. Menon, and K. Soman, “Nse stock market prediction using deep-learning models,” *Procedia computer science*, vol. 132, pp. 1351–1362, 2018.
- [64] Y. Zhao, X. Jing, K. Qian, D. F. Gomes, and S. Luo, “Skill generalization of tubular object manipulation with tactile sensing and sim2real learning,” *Robotics and Autonomous Systems*, vol. 160, p. 104321, 2023.
- [65] P.-F. Tsai, C.-H. Liao, and S.-M. Yuan, “Using deep learning with thermal imaging for human detection in heavy smoke scenarios,” *Sensors*, vol. 22, no. 14, p. 5351, 2022.
- [66] H. Face and H. Face, “The ai community building the future,” *URL: <https://huggingface.co>*, 2021.
- [67] SparkNLP, “Sparknlp,” *<https://sparknlp.org/>*, 2021.

- [68] W. Jiang, N. Synovic, R. Sethi, A. Indarapu, M. Hyatt, T. R. Schorlemmer, G. K. Thiruvathukal, and J. C. Davis, “An empirical study of artifacts and security risks in the pre-trained model supply chain,” in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 2022, pp. 105–114.
- [69] S. Wang, Y. Zhao, X. Hou, and H. Wang, “Large language model supply chain: A research agenda,” *arXiv preprint arXiv:2404.12736*, 2024.
- [70] J. Zhao, S. Wang, Y. Zhao, X. Hou, K. Wang, P. Gao, Y. Zhang, C. Wei, and H. Wang, “Models are codes: Towards measuring malicious code poisoning attacks on pre-trained model hubs,” *arXiv preprint arXiv:2409.09368*, 2024.
- [71] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: <https://www.tensorflow.org/>
- [72] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.
- [73] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv preprint arXiv:1512.01274*, 2015.
- [74] R. R. Curtin, M. Edel, O. Shrit, S. Agrawal, S. Basak, J. J. Balamuta, R. Birmingham, K. Dutt, D. Eddelbuettel, R. Garg *et al.*, “mlpack 4: a fast, header-only c++ machine learning library,” *arXiv preprint arXiv:2302.00820*, 2023.
- [75] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

- [76] “Nvidia® tensorrt™, an sdk for high-performance deep learning inference, includes a deep learning inference optimizer and runtime that delivers low latency and high throughput for inference applications.” [Online]. Available: <https://developer.nvidia.com/tensorrt>
- [77] “Production-grade ai engine to speed up training and inferencing in your existing technology stack.” [Online]. Available: <https://onnxruntime.ai/docs/performance/graph-optimizations.html>
- [78] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, “{TVM}: An automated {End-to-End} optimizing compiler for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.
- [79] NVIDIA, P. Vingelmann, and F. H. Fitzek, “Cuda, release: 10.2.89,” 2020. [Online]. Available: <https://developer.nvidia.com/cuda-toolkit>
- [80] J. So, J. Hwangbo, S. H. Kim, and I. Yun, “Analysis on autonomous vehicle detection performance according to various road geometry settings,” *Journal of Intelligent Transportation Systems*, vol. 27, no. 3, pp. 384–395, 2023.
- [81] G. Algan and I. Ulusoy, “Image classification with deep learning in the presence of noisy labels: A survey,” *Knowledge-Based Systems*, vol. 215, p. 106771, 2021.
- [82] J. Yu, Z. Wang, V. Vasudevan, L. Yeung, M. Seyedhosseini, and Y. Wu, “Coca: Contrastive captioners are image-text foundation models,” *arXiv preprint arXiv:2205.01917*, 2022.
- [83] R. C. Çalik and M. F. Demirci, “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, 2018, pp. 1–2.
- [84] J. Wang, Z. Yang, X. Hu, L. Li, K. Lin, Z. Gan, Z. Liu, C. Liu, and L. Wang, “Git: A generative image-to-text transformer for vision and language,” *arXiv preprint arXiv:2205.14100*, 2022.
- [85] X. Hu, Z. Gan, J. Wang, Z. Yang, Z. Liu, Y. Lu, and L. Wang, “Scaling up vision-language pre-training for image captioning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 17 980–17 989.

- [86] M. B. Sariyildiz, J. Perez, and D. Larlus, “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, 2020, pp. 153–170.
- [87] J. Chen, H. Guo, K. Yi, B. Li, and M. Elhoseiny, “Visualgpt: Data-efficient adaptation of pretrained language models for image captioning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 18 030–18 040.
- [88] S. Minaee and Z. Liu, “Automatic question-answering using a deep similarity neural network,” in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2017, pp. 923–927.
- [89] B.-D. Oh and W. Schuler, “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.
- [90] L. Gessler and A. Zeldes, “Microbert: Effective training of low-resource monolingual berts through parameter reduction and multitask learning,” *arXiv preprint arXiv:2212.12510*, 2022.
- [91] Y. Tang, X. Cheng, and W. Lu, “Improving complex knowledge base question answering via question-to-action and question-to-question alignment,” *arXiv preprint arXiv:2212.13036*, 2022.
- [92] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pre-training approach,” *arXiv preprint arXiv:1907.11692*, 2019.
- [93] H. Liu, K. Simonyan, and Y. Yang, “Darts: Differentiable architecture search,” *arXiv preprint arXiv:1806.09055*, 2018.
- [94] J. Howard and S. Ruder, “Universal language model fine-tuning for text classification,” *arXiv preprint arXiv:1801.06146*, 2018.
- [95] A. M. Dai and Q. V. Le, “Semi-supervised sequence learning,” *Advances in neural information processing systems*, vol. 28, 2015.
- [96] T. Beltramelli, “pix2code: Generating code from a graphical user interface screenshot,” in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2018, pp. 1–6.

- [97] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [98] P. Yin and G. Neubig, “Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation,” *arXiv preprint arXiv:1810.02720*, 2018.
- [99] M. Yasunaga and P. Liang, “Graph-based, self-supervised program repair from diagnostic feedback,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 10 799–10 808.
- [100] F. Mahdisoltani, G. Berger, W. Gharbieh, D. Fleet, and R. Memisevic, “Fine-grained video classification and captioning,” *arXiv preprint arXiv:1804.09235*, vol. 5, no. 6, 2018.
- [101] R. Patgiri, “A taxonomy on big data: Survey,” *arXiv preprint arXiv:1808.08474*, 2018.
- [102] Y. Lv, B. Liu, J. Zhang, Y. Dai, A. Li, and T. Zhang, “Semi-supervised active salient object detection,” *Pattern Recognition*, vol. 123, p. 108364, 2022.
- [103] R. Simhambhatla, K. Okiah, S. Kuchkula, and R. Slater, “Self-driving cars: Evaluation of deep learning techniques for object detection in different driving conditions,” *SMU Data Science Review*, vol. 2, no. 1, p. 23, 2019.
- [104] S. Ramos, S. Gehrig, P. Pinggera, U. Franke, and C. Rother, “Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling,” in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 1025–1032.
- [105] R. Kulkarni, S. Dhavalikar, and S. Bangar, “Traffic light detection and recognition for self driving cars using deep learning,” in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–4.
- [106] R. G. Athreya, S. K. Bansal, A.-C. N. Ngomo, and R. Usbeck, “Template-based question answering using recursive neural networks,” in *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*. IEEE, 2021, pp. 195–198.

- [107] P. K. Roy, “Deep neural network to predict answer votes on community question answering sites,” *Neural Processing Letters*, vol. 53, no. 2, pp. 1633–1646, 2021.
- [108] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, “Towards deep learning models resistant to adversarial attacks,” *arXiv preprint arXiv:1706.06083*, 2017.
- [109] S. McGregor, “Preventing repeated real world ai failures by cataloging incidents: The ai incident database,” 2020.
- [110] F. Thung, S. Wang, D. Lo, and L. Jiang, “An empirical study of bugs in machine learning systems,” in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 271–280.
- [111] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, “The symptoms, causes, and repairs of bugs inside a deep learning library,” *Journal of Systems and Software*, vol. 177, p. 110935, 2021.
- [112] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.
- [113] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, “An empirical study on tensorflow program bugs,” in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 129–140.
- [114] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, “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*, 2021, pp. 968–980.
- [115] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, “A comprehensive study of autonomous vehicle bugs,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 385–396.
- [116] A. Di Franco, H. Guo, and C. Rubio-González, “A comprehensive study of real-world numerical bug characteristics,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 509–519.

- [117] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: a system for large-scale machine learning.” in *Osdi*, vol. 16, no. 2016. Savannah, GA, USA, 2016, pp. 265–283.
- [118] M. L. Y. M. S. G. S. Z. R.R. Curtin, M. Edel, “mlpack 3: a fast, flexible c++ machine learning library,” p. 726, 2018.
- [119] W. McKinney *et al.*, “Data structures for statistical computing in python,” in *Proceedings of the 9th Python in Science Conference*, vol. 445. Austin, TX, 2010, pp. 51–56.
- [120] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, “Array programming with numpy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.
- [121] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020.
- [122] Y. Zhou and A. Sharma, “Automated identification of security issues from commit messages and bug reports,” in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 914–919.
- [123] W. Al-Kahla, A. S. Shatnawi, and E. Taqieddin, “A taxonomy of web security vulnerabilities,” in *2021 12th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2021, pp. 424–429.
- [124] P. Sharma and J. Singh, “Systematic literature review on software effort estimation using machine learning approaches,” in *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*. IEEE, 2017, pp. 43–47.
- [125] T. Aslam, “A taxonomy of security faults in the unix operating system,” *Master’s thesis, Purdue University*, vol. 199, no. 5, 1995.

- [126] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug characteristics in open source software,” *Empirical software engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.
- [127] A. Romano, X. Liu, Y. Kwon, and W. Wang, “An empirical study of bugs in webassembly compilers,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 42–54.
- [128] T. Hirsch and B. Hofer, “Root cause prediction based on bug reports,” in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 171–176.
- [129] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, “A systematic evaluation of static api-misuse detectors,” *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.
- [130] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, “Identifying the characteristics of vulnerable code changes: An empirical study,” in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 257–268.
- [131] C. Q. Adamsen, A. Møller, S. Alimadadi, and F. Tip, “Practical ajax race detection for javascript web applications,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 38–48.
- [132] E. Arteca, S. Harner, M. Pradel, and F. Tip, “Nessie: Automatically testing javascript apis with asynchronous callbacks.” ICSE, 2022.
- [133] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, “Dependency smells in javascript projects,” *IEEE Transactions on Software Engineering*, 2021.
- [134] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, “An empirical study of adoption of software testing in open source projects,” in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 103–112.
- [135] Q. Xiao, K. Li, D. Zhang, and W. Xu, “Security risks in deep learning implementations,” in *2018 IEEE Security and privacy workshops (SPW)*. IEEE, 2018, pp. 123–128.

- [136] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, “Vuldeelocator: a deep learning-based fine-grained vulnerability detector,” *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [137] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: Using static analysis to find bugs in the real world,” *Commun. ACM*, vol. 53, no. 2, p. 66–75, feb 2010. [Online]. Available: <https://doi.org/10.1145/1646353.1646374>
- [138] G. inc. Errorprone. [Online]. Available: <https://errorprone.info/>
- [139] Facebook. (2013) Infer. [Online]. Available: <https://fbinfer.com/>
- [140] SpotBugs. (2021) Spotbugs. [Online]. Available: <https://spotbugs.github.io/>
- [141] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, “Using static analysis to find bugs,” *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.
- [142] D. A. Wheeler. (2013) Dlawfinder. [Online]. Available: <http://dwheeler.com/flawfinder/>
- [143] J. Chen, C. Zhang, S. Cai, L. Zhang, and L. Ma, “A memory-related vulnerability detection approach based on vulnerability model with petri net,” *Journal of Logical and Algebraic Methods in Programming*, vol. 132, p. 100859, 2023.
- [144] J. D. Pereira and M. Vieira, “On the use of open-source c/c++ static analysis tools in large projects,” in *2020 16th European Dependable Computing Conference (EDCC)*. IEEE, 2020, pp. 97–102.
- [145] C. Mitropoulos, “Employing different program analysis methods to study bug evolution,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1202–1204.
- [146] M. Gu, H. Feng, H. Sun, P. Liu, Q. Yue, J. Hu, C. Cao, and Y. Zhang, “Hierarchical attention network for interpretable and fine-grained vulnerability detection,” in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2022, pp. 1–6.
- [147] D. Zou, Y. Hu, W. Li, Y. Wu, H. Zhao, and H. Jin, “mvulpreter: A multi-granularity vulnerability detection system with interpretations,” *IEEE Transactions on Dependable and Secure Computing*, 2022.

- [148] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *arXiv preprint arXiv:1909.03496*, 2019.
- [149] A. Dunham, “flawfinder,” <https://github.com/andrew-d/rough-auditing-tool-for-security>, 2009.
- [150] D. Marjamäki. (2016) Cppcheck. [Online]. Available: <https://cppcheck.sourceforge.io/>
- [151] S. Pujar, Y. Zheng, L. Buratti, B. Lewis, A. Morari, J. Laredo, K. Postlethwait, and C. Görn, “Varangian: a git bot for augmented static analysis,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 766–767.
- [152] S. Mehrpour and T. D. LaToza, “Can static analysis tools find more defects? a qualitative study of design rule violations found by code review,” *Empirical Software Engineering*, vol. 28, no. 1, p. 5, 2023.
- [153] C. Lattner, “Llvm and clang: Next generation compiler technology,” in *The BSD conference*, vol. 5, 2008, pp. 1–20.
- [154] K. Umann and Z. Porkoláb, “Detecting uninitialized variables in c++ with the clang static analyzer,” *Acta Cybernetica*, vol. 25, no. 4, pp. 923–940, 2022.
- [155] P. G. Szécsi, G. Horváth, and Z. Porkoláb, “Improved loop execution modeling in the clang static analyzer,” *Acta Cybernetica*, vol. 25, no. 4, pp. 909–921, 2022.
- [156] H. Aslanyan, Z. Gevorgyan, R. Mkoyan, H. Movsisyan, V. Sahakyan, and S. Sargsyan, “Static analysis methods for memory leak detection: A survey,” in *2022 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 2022, pp. 1–6.
- [157] M. Pradel and T. R. Gross, “Detecting anomalies in the order of equally-typed method arguments,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 232–242.
- [158] Y. Younan, W. Joosen, and F. Piessens, “Runtime countermeasures for code injection attacks against c and c++ programs,” *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, pp. 1–28, 2012.

- [159] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” *arXiv preprint arXiv:1801.01681*, 2018.
- [160] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, “Vulsniper: Focus your attention to shoot fine-grained vulnerabilities.” in *IJCAI*, 2019, pp. 4665–4671.
- [161] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, “On the naturalness of software,” *Commun. ACM*, vol. 59, no. 5, p. 122–131, apr 2016. [Online]. Available: <https://doi.org/10.1145/2902362>
- [162] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.
- [163] D. A. Tomassi, “Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 980–982.
- [164] N. Rutar, C. B. Almazan, and J. S. Foster, “A comparison of bug finding tools for java,” in *15th International symposium on software reliability engineering*. IEEE, 2004, pp. 245–256.
- [165] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: using static analysis to find bugs in the real world,” *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.
- [166] N. Ayewah and W. Pugh, “The google findbugs fixit,” in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 241–252.
- [167] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating systems errors,” in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.
- [168] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

- [169] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, “An empirical study of client-side javascript bugs,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 55–64.
- [170] M. Selakovic and M. Pradel, “Performance issues and optimizations in javascript: an empirical study,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 61–72.
- [171] J. Shi, Y. Xiao, Y. Li, Y. Li, D. Yu, C. Yu, H. Su, Y. Chen, and W. Huo, “Acetest: Automated constraint extraction for testing deep learning operators,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 690–702.
- [172] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, “Fuzzing automatic differentiation in deep-learning libraries,” *arXiv preprint arXiv:2302.04351*, 2023.
- [173] E. First, M. N. Rabe, T. Ringer, and Y. Brun, “Baldur: whole-proof generation and repair with large language models,” *arXiv preprint arXiv:2303.04910*, 2023.
- [174] Y. Wei, C. S. Xia, and L. Zhang, “Copiloting the copilots: Fusing large language models with completion engines for automated program repair,” *arXiv preprint arXiv:2309.00608*, 2023.
- [175] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, “Multilingual code co-evolution using large language models,” *arXiv preprint arXiv:2307.14991*, 2023.
- [176] J. Ba, G. J. Duck, and A. Roychoudhury, “Efficient greybox fuzzing to detect memory errors,” in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
- [177] A. Kallingal Joshy and W. Le, “Fuzzeraid: Grouping fuzzed crashes based on fault signatures,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
- [178] H. Lee, S. Kim, and S. K. Cha, “Fuzzle: Making a puzzle for fuzzers,” in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
- [179] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, “Griffin: Grammar-free dbms fuzzing,” in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

- [180] Y. Yu, X. Jia, Y. Liu, Y. Wang, Q. Sang, C. Zhang, and P. Su, “Htfuzz: Heap operation sequence sensitive fuzzing,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
- [181] S. Groß. (2019) coverage-guided fuzzer for dynamic language interpreters based on a custom intermediate language. [Online]. Available: <https://github.com/googleprojectzero/fuzzilli>
- [182] J. Gu, X. Luo, Y. Zhou, and X. Wang, “Muffin: Testing deep learning libraries via neural architecture fuzzing,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1418–1430.
- [183] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, “Finding deep-learning compilation bugs with nnsmith,” *arXiv preprint arXiv:2207.13066*, 2022.
- [184] Q. Su, C. Geng, G. Pekhimenko, and X. Si, “Torchprobe: Fuzzing dynamic deep learning compilers,” *arXiv preprint arXiv:2310.20078*, 2023.
- [185] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, “Coverage-guided tensor compiler fuzzing with joint ir-pass mutation,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–26, 2022.
- [186] W. Luo, D. Chai, X. Ruan, J. Wang, C. Fang, and Z. Chen, “Graph-based fuzz testing for deep learning inference engines,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 288–299.
- [187] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.
- [188] B. Van Merriënboer, O. Breuleux, A. Bergeron, and P. Lamblin, “Automatic differentiation in ml: Where we are and where we should be going,” *Advances in neural information processing systems*, vol. 31, 2018.
- [189] J. Vermorel and M. Mohri, “Multi-armed bandit algorithms and empirical evaluation,” in *European conference on machine learning*. Springer, 2005, pp. 437–448.

- [190] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.
- [191] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, “Deep learning library testing via effective model generation,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 788–799. [Online]. Available: <https://doi.org/10.1145/3368089.3409761>
- [192] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, “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, 2019, pp. 1027–1038.
- [193] Y. Li, S. Wang, and T. N. Nguyen, “Dear: A novel deep learning-based approach for automated program repair,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.
- [194] F. Ertam and G. Aydin, “Data classification with deep learning using tensorflow,” in *2017 international conference on computer science and engineering (UBMK)*. IEEE, 2017, pp. 755–758.
- [195] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, “Automatic static vulnerability detection for machine learning libraries: Are we there yet?” in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 795–806.
- [196] Y. Ishida, T. Shimizu, and M. Yoshikawa, “An analysis and comparison of keyword recommendation methods for scientific data,” *International Journal on Digital Libraries*, vol. 21, no. 3, pp. 307–327, 2020.
- [197] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, “Toward understanding deep learning framework bugs,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, 2023.
- [198] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

- [199] X. Han and T. Yu, “An empirical study on performance bugs for highly configurable software systems,” in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–10.
- [200] M. Wei, Y. Huang, J. Wang, J. Shin, N. S. Harzevili, and S. Wang, “Api recommendation for machine learning libraries: how far are we?” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 370–381.
- [201] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, “Focus: A recommender system for mining api function calls and usage patterns,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.
- [202] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [203] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “Mapo: Mining and recommending api usage patterns,” in *ECOOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. Springer, 2009, pp. 318–343.
- [204] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, “Mining succinct and high-coverage api usage patterns from source code,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 319–328.
- [205] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, “Nnsmith: Generating diverse and valid test cases for deep learning compilers,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 530–543.
- [206] M. Zitser, R. Lippmann, and T. Leek, “Testing static analysis tools using exploitable buffer overflows from open source code,” in *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004, pp. 97–106.

- [207] G. Chatzieleftheriou and P. Katsaros, “Test-driving static analysis tools in search of c code vulnerabilities,” in *2011 IEEE 35th annual computer software and applications conference workshops*. IEEE, 2011, pp. 96–103.
- [208] I. X. Gauthier, M. Lamothe, G. Mussbacher, and S. McIntosh, “Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 30–41.
- [209] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. R. Lyu, “Revisiting, benchmarking and exploring api recommendation: How far are we?” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1876–1897, 2022.
- [210] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, “Automatic unit test generation for machine learning libraries: How far are we?” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1548–1560.
- [211] A. Hazimeh, A. Herrera, and M. Payer, “Magma: A ground-truth fuzzing benchmark,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.
- [212] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, “Fuzzbench: an open fuzzer benchmarking platform and service,” in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1393–1403.
- [213] R. Natella and V.-T. Pham, “Profuzzbench: A benchmark for stateful protocol fuzzing,” in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 662–665.
- [214] M. Böhme, L. Szekeres, and J. Metzman, “On the reliability of coverage-based fuzzer benchmarking,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1621–1633.
- [215] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer, “Igor: Crash deduplication through root-cause clustering,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3318–3336.

- [216] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.
- [217] J. Liu, Y. Huang, Z. Wang, L. Ma, C. Fang, M. Gu, X. Zhang, and Z. Chen, “Generation-based differential fuzzing for deep learning libraries,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–28, 2023.
- [218] J. Wang, T. Lutellier, S. Qian, H. V. Pham, and L. Tan, “Eagle: creating equivalent graphs to test deep learning libraries,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 798–810.
- [219] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, “Audee: Automated testing for deep learning frameworks,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 486–498.
- [220] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, “Deep learning library testing via effective model generation,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 788–799.
- [221] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, “{IvySyn}: Automated vulnerability discovery in deep learning frameworks,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2383–2400.
- [222] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, “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*, 2018, pp. 132–142.
- [223] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, “Deepbillboard: Systematic physical-world testing of autonomous driving systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 347–358.
- [224] J. Zhang, Q. Qin, Q. Ye, and T. Ruan, “St-unet: Swin transformer boosted u-net with cross-layer feature enhancement for medical image segmentation,” *Computers in Biology and Medicine*, p. 106516, 2023.

- [225] P. Benedetti, M. Femminella, and G. Reali, “Mixed-sized biomedical image segmentation based on u-net architectures,” *Applied Sciences*, vol. 13, no. 1, p. 329, 2023.
- [226] Z. Fang, X. Ma, H. Pan, G. Yang, and G. R. Arce, “Movement forecasting of financial time series based on adaptive lstm-bn network,” *Expert Systems with Applications*, vol. 213, p. 119207, 2023.
- [227] Y. Ji, X. Zhang, S. Ji, X. Luo, and T. Wang, “Model-reuse attacks on deep learning systems,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 349–363.
- [228] T. Byun and S. Rayadurgam, “Manifold for machine learning assurance,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 97–100.
- [229] C. Barrett, B. Boyd, E. Burzstein, N. Carlini, B. Chen, J. Choi, A. R. Chowdhury, M. Christodorescu, A. Datta, S. Feizi *et al.*, “Identifying and mitigating the security risks of generative ai,” *arXiv preprint arXiv:2308.14840*, 2023.
- [230] N. Carlini, “A llm assisted exploitation of ai-guardian,” *arXiv preprint arXiv:2307.15008*, 2023.
- [231] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, “Automatic testing and improvement of machine translation,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 974–985.
- [232] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, “Input prioritization for testing neural networks,” in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 2019, pp. 63–70.
- [233] D. Cofer, I. Amundson, R. Sattigeri, A. Passi, C. Boggs, E. Smith, L. Gilham, T. Byun, and S. Rayadurgam, “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, 2020, pp. 361–368.
- [234] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “Coconut: combining context-aware neural translation models using ensemble for program repair,” in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

- [235] S. Wang and N. Nagappan, “Characterizing and understanding software developer networks in security development,” in *The 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)*, 2021.
- [236] Y. Tian, J. Lawall, and D. Lo, “Identifying linux bug fixing patches,” in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 386–396.
- [237] “History-driven fuzzing for deep learning libraries,” 2023. [Online]. Available: <https://github.com/dmc1778/Orion>
- [238] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, “History-guided configuration diversification for compiler test-program generation,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 305–316.
- [239] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, “History-driven test program synthesis for jvm testing,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1133–1144.
- [240] J.-M. Zhang, Z.-Q. Cui, X. Chen, H.-H. Wu, L.-W. Zheng, and J.-B. Liu, “Deltafuzz: historical version information guided fuzz testing,” *Journal of Computer Science and Technology*, vol. 37, no. 1, pp. 29–49, 2022.
- [241] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.
- [242] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated whitebox testing of deep learning systems,” in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.
- [243] A. Graese, A. Rozsa, and T. E. Boult, “Assessing threat of adversarial examples on deep neural networks,” in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2016, pp. 69–74.
- [244] A. Agarwal, N. Ratha, M. Vatsa, and R. Singh, “Exploring robustness connection between artificial and natural adversarial examples,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 179–186.
- [245] T. Dai, Y. Feng, B. Chen, J. Lu, and S.-T. Xia, “Deep image prior based defense against adversarial examples,” *Pattern Recognition*, vol. 122, p. 108249, 2022.

- [246] S.-H. Choi, J.-M. Shin, P. Liu, and Y.-H. Choi, “Argan: Adversarially robust generative adversarial networks for deep neural networks against adversarial examples,” *IEEE Access*, vol. 10, pp. 33 602–33 615, 2022.
- [247] J. Zhang and C. Li, “Adversarial examples: Opportunities and challenges,” *IEEE transactions on neural networks and learning systems*, vol. 31, no. 7, pp. 2578–2593, 2019.
- [248] J. Bose, G. Gidel, H. Berard, A. Cianflone, P. Vincent, S. Lacoste-Julien, and W. Hamilton, “Adversarial example games,” *Advances in neural information processing systems*, vol. 33, pp. 8921–8934, 2020.
- [249] H. J. Kang, P. Rattanukul, S. A. Haryono, T. G. Nguyen, C. Ragkhitwetsagul, C. Pasareanu, and D. Lo, “Skipfuzz: Active learning-based input selection for fuzzing deep learning libraries,” *arXiv preprint arXiv:2212.04038*, 2022.
- [250] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, “{SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 479–494.
- [251] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, “Cross-checking semantic correctness: The case of finding file system bugs,” in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 361–377.
- [252] L. Situ, L. Wang, Y. Liu, B. Mao, and X. Li, “Vanguard: Detecting missing checks for prognosing potential vulnerabilities,” in *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, 2018, pp. 1–10.
- [253] W. Wang, K. Lu, and P.-C. Yew, “Check it again: Detecting lacking-recheck bugs in os kernels,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1899–1913.
- [254] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 499–510.
- [255] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, “Autoises: Automatically inferring security specification and detecting violations.” in *USENIX Security Symposium*, 2008, pp. 379–394.

- [256] S. Son, K. S. McKinley, and V. Shmatikov, “Rolecast: finding missing security checks when you do not know what checks are,” in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 1069–1084.
- [257] S. Jana, Y. J. Kang, S. Roth, and B. Ray, “Automatically detecting error handling bugs using error specifications,” in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 345–362.
- [258] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via chatgpt,” *arXiv preprint arXiv:2304.07590*, 2023.
- [259] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, “Chatgpt for vulnerability detection, classification, and repair: How far are we?” in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 632–636.
- [260] S. Zeng, J. Zhang, P. He, Y. Xing, Y. Liu, H. Xu, J. Ren, S. Wang, D. Yin, Y. Chang *et al.*, “The good and the bad: Exploring privacy issues in retrieval-augmented generation (rag),” *arXiv preprint arXiv:2402.16893*, 2024.
- [261] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.
- [262] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov *et al.*, “Theano: A python framework for fast computation of mathematical expressions,” *arXiv preprint arXiv:1605.02688*, 2016.
- [263] F. Seide and A. Agarwal, “Cntk: Microsoft’s open-source deep-learning toolkit,” in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 2135–2135.
- [264] K. Huang, B. Chen, S. Wu, J. Cao, L. Ma, and X. Peng, “Demystifying dependency bugs in deep learning stack,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 450–462.
- [265] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, “A survey on evaluation of large language models,”

ACM Transactions on Intelligent Systems and Technology, vol. 15, no. 3, pp. 1–45, 2024.

- [266] M. M. Mohajer, R. Aleithan, N. S. Harzevili, M. Wei, A. B. Belle, H. V. Pham, and S. Wang, “Effectiveness of chatgpt for static analysis: How far are we?” in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024, pp. 151–160.
- [267] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software testing with large language models: Survey, landscape, and vision,” *IEEE Transactions on Software Engineering*, 2024.
- [268] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, “Clarifygpt: Empowering llm-based code generation with intention clarification,” *arXiv preprint arXiv:2310.10996*, 2023.
- [269] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen *et al.*, “A comprehensive capability analysis of gpt-3 and gpt-3.5 series models,” *arXiv preprint arXiv:2303.10420*, 2023.
- [270] P. Finardi, L. Avila, R. Castaldoni, P. Gengo, C. Larcher, M. Piau, P. Costa, and V. Caridá, “The chronicles of rag: The retriever, the chunk and the generator,” *arXiv preprint arXiv:2401.07883*, 2024.
- [271] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried, “Coderag-bench: Can retrieval augment code generation?” *arXiv preprint arXiv:2406.14497*, 2024.
- [272] N. Ampazis, “Improving rag quality for large language models with topic-enhanced reranking,” in *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 2024, pp. 74–87.
- [273] E. Vergou, I. Pagouni, M. Nanos, and K. L. Kermanidis, “Readability classification with wikipedia data and all-minilm embeddings,” in *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 2023, pp. 369–380.
- [274] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

- [275] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.
- [276] J. Chen, Y. Geng, Z. Chen, J. Z. Pan, Y. He, W. Zhang, I. Horrocks, and H. Chen, “Zero-shot and few-shot learning with knowledge graphs: A comprehensive survey,” *Proceedings of the IEEE*, vol. 111, no. 6, pp. 653–685, 2023.
- [277] C. Pornprasit and C. Tantithamthavorn, “Fine-tuning and prompt engineering for large language models-based code review automation,” *Information and Software Technology*, p. 107523, 2024.
- [278] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, “Llm is like a box of chocolates: the non-determinism of chatgpt in code generation,” *arXiv preprint arXiv:2308.02828*, 2023.
- [279] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, “Autocoderover: Autonomous program improvement,” *arXiv preprint arXiv:2404.05427*, 2024.
- [280] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “Swe-bench: Can language models resolve real-world github issues?” *arXiv preprint arXiv:2310.06770*, 2023.
- [281] N. S. Harzevili, M. M. Mohajer, M. Wei, H. V. Pham, and S. Wang, “Security knowledge-guided fuzzing of deep learning libraries,” 2023.
- [282] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.
- [283] LLVMASAN, “LLVMASAN,” <https://clang.llvm.org/docs/AddressSanitizer.html>.
- [284] googleASAN, “googleASAN,” <https://github.com/google/sanitizers/wiki/AddressSanitizer>.
- [285] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu, “Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing.” in *USENIX Security Symposium*, 2019, pp. 425–442.
- [286] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, “A3test: Assertion-augmented automated test case generation,” *arXiv preprint arXiv:2302.10352*, 2023.

- [287] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, “Generating accurate assert statements for unit test cases using pretrained transformers,” in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 54–64.
- [288] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhi-
mandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou,
B. Natarajan, and M. Daga, “Miopen: An open source library for deep learning
primitives,” 2019.