Empirical Software Engineering (2024) 29:45
https://doi.org/10.1007/s10664-023-10413-9

Demystifying API misuses in deep learning applications

Deheng Yang[1] · Kui Liu[2] · Yan Lei[3,4] · Li Li[5] · Huan Xie[3,4] · Chunyan Liu[3,4] · Zhenyu Wang[3] · Xiaoguang Mao[1] · Tegawendé F. Bissyandé[6,7]

Accepted: 18 October 2023 / Published online: 16 February 2024

Abstract
Deep Learning (DL) is achieving staggering performance on an increasing number of applications in various areas. Meanwhile, its associated data-driven programming paradigm comes with a set of challenges for the software engineering community, including the debugging activities for DL applications. Recent empirical studies on bugs in DL applications have shown that the API (i.e., Application Program Interface) misuse has been flagged as an important category of DL programming bugs. By exploring this literature towards API misuse bugs in DL applications, we identified three barriers that are locking an entire research direction. However, three barriers are hindering progress in this research direction: misclassification of API misuse bugs, lack of relevant dataset, and limited depth of analysis. Our work unlocks these barriers by providing an in-depth analysis of a frequent bug type that appears as a mystery. Concretely, we first offer a new perspective to a significant misclassification issue in the literature that hinders understanding of API misuses in DL applications. Subsequently, we curate the first dataset MISUAPI of 143 API misuses sampled from real-world DL applications. Finally, we perform systematic analyses to dissect API misuses and enumerate the symptoms of API misuses in DL applications as well as investigate the possibility of detecting them with state-of-the-art static analyzers. Overall, the insights summarized in this work are important for the community: 1) 18-35% of real API misuses are mislabelled in existing DL bug studies; 2) the widely adopted API misuse taxonomy, namely MUC, does not cover the cases of 1 out of 3 encountered API misuses; 3) DL library API misuses show significant differences from the general third-party library API misuses in terms of the API-usage element issue and symptoms; 4) Most (92.3%) API misuses lead to program crashes; 5) 95.8% API misuses remain undetectable by state-of-the-art static analyzers.

Keywords API misuse · Deep learning application · Bug dataset · Empirical study

Communicated by: Denys Poshyvanyk

✉ Yan Lei
yanlei@cqu.edu.cn

Extended author information available on the last page of the article

1 Introduction

While "software is eating the world", deep learning (DL) is rapidly swallowing software engineering in various application areas CEO Nvidia (2023), ranging from futuristic (e.g., code generation) Yang et al. (2020); Cambronero et al. (2019); Svyatkovskiy et al. (2020), safety-critical (e.g., self-driving cars Kuutti et al. 2020 or medical diagnosis Liu et al. 2019) to simple leisure (e.g., gaming) applications. Meanwhile, the new programming paradigm of DL, which trains problem-solving models with massive data rather than handcrafting decision-making programs, has brought new debugging challenges to the software community Dilhara et al. (2021); Meijer (2018). Towards enabling the emergence of novel approaches to address the requirements of DL development and debugging, several empirical studies (Zhang et al. 2018; Humbatova et al. 2020; Zhang et al. 2020; Chen et al. 2021; Islam et al. 2019a; Zhang et al. 2019; Islam et al. 2020) have been conducted in recent years to characterize bugs in DL applications. Across all such studies, *API misuse* is presented as a common and important category of bugs in DL applications. In a more recent survey, Lamothe et al. (2021) reported that API misuse is one of the three major API evolution challenges for practitioners. Therefore, there is a need to deepen the community knowledge of API misuses in DL applications to boost the debugging momentum of these bugs in the ever-increasing portion of software that relies on DL.

However, by exploring the literature on API misuses in DL applications, we identify three barriers that impede the comprehensive understanding of practitioners and the development of automated techniques towards API misuses in DL applications. The first barrier relates to the inconsistent definitions of API misuses in the literature targeting DL applications. After a systematic literature review on empirical studies that cover API misuses in DL applications (listed in Table 1), we observed that these studies present different or even obscure definitions on API misuses. For example, API incompatibility/update issues are also considered as API misuses (Zhang et al. 2020; Islam 2020). Yet, at least one previous study Zhang et al. (2018) on TensorFlow-based program bugs has placed such in the category of API change bugs. Such inconsistencies in the fundamental definition increase the difficulty of an in-depth study on API misuses in DL applications. Therefore, we propose to remove the first barrier by investigating the following research question:

RQ1: how can API misuses be defined to ensure that they can be differentiated from other bugs occurring in DL applications?

The second barrier is characterized as the limitation of current API misuses datasets in DL applications to support future research. To build knowledge, one requires reliable data. However, despite increasing attention in the community Wan et al. (2021), such data is missing regarding API misuses in DL applications. Prior empirical studies did not publicly release their datasets (e.g., Zhang et al. 2020; Wan et al. 2021), investigated a relatively small number of API misuse cases (e.g., 33 in Zhang et al. 2018), or mixed API misuses within other bug datasets (e.g., Zhang et al. 2018; Humbatova et al. 2020). In particular, existing publicly available datasets contain at most 33 API misuses (i.e., 33 Zhang et al. 2018) in DL applications. Such scale of dataset still has limitations in supporting a further study on API misuses in DL applications. We thus propose to resolve a second barrier by answering the question:

RQ2: where can we collect a significant number of API misuse samples from real-world DL applications to publicly release for the community?

The third barrier corresponds to a limited depth in the analyses of API misuses in DL applications. API misuses in traditional software code have seen in-depth studies in the literature

Table 1 The reviewed literature that involves study of API-related bugs in DL applications

Literature	Year	Venue	Dataset	API misuse description
Zhang et al. (2018)	2018	ISSTA	Available	33 (18.9%) API misuses out of 175 bugs in Tensorflow projects are identified.
Humbatova et al. (2020)	2020	ICSE	Available	For API bugs, the most frequent is wrong API usage (i.e., API misuse).
Islam et al. (2019b)	2019	arXiv	Unavailable	–
Chen et al. (2021)	2019	ICSE	Available but not pertinent	Developers often misuse relevant APIs provided by DL frameworks.
Zhang et al. (2019)	2019	ISSRE	Unavailable	API misuse is one of the 5 main root causes of DL programming issues.
Islam et al. (2019a)	2019	FSE	Unavailable	Most of the non model related bugs are caused by API Misuse (6% - 100%).
Zhang et al. (2020)	2020	ICSE	Unavailable	138 (20.7%) framework API misuses out of 668 DL-specific bugs are identified.
Wu et al. (2021)	2021	arXiv	Unavailable	–
Islam et al. (2020)	2020	ICSE	Available	–

(Amann et al. 2016, 2018, 2019; Kechagia et al. 2021) towards proposing widely adopted taxonomies (e.g., Amann et al. 2018 identified 14 categories of API misuses). However, API misuse for DL is different from traditional applications for the following three aspects: (1) many DL frameworks, such as TensorFlow and PyTorch, are implemented in dynamically typed languages, i.e., Python. This makes it harder to catch errors during the development process, as errors related to parameter types are not detected until runtime. This can lead to more frequent API misuses in DL applications compared to traditional applications. (2) DL applications often deal with large amounts of data, which can be challenging to manage and process correctly. This can lead to API misuses related to data handling, such as incorrect data types or shapes being passed to an API. (3) DL applications rely heavily on specialized libraries and APIs that may not be widely used in traditional applications. This means that developers may not have as much experience or knowledge of these libraries, which can lead to more frequent API misuses. In other words, API misuses in DL applications have been superficially approached, where the literature still needs to close the gap in understanding the specificities of such bugs in the context of DL. For example, questions on whether API misuses in DL applications can be detected by existing analysis tools for traditional programs remain unanswered. Thus, we propose to take up the challenge of tackling the question:

RQ3: what are the specific characteristics of API misuses in DL applications?

The aforementioned three barriers could impede the understanding of DL application practitioners on resolving such API misuses in development activities. The aforementioned three barriers could impede the understanding of DL application practitioners on resolving such API misuses in development activities, and they could also block the prototyping of automated detection or repair tools for API misuses in DL applications. The API specification (i.e., API usage contract) needed by API misuse repair techniques cannot be directly obtained from the API documentation, as API documentation is typically written in natural language and often implicitly encodes part of but not complete API usage contract (e.g., raising an Exception when the parameter is null). In this way, it is difficult to construct repair tools from API specifications and documentation. This also explains why existing API misuse detection or repair tools for traditional applications are constructed by collecting examples (Amann et al. 2018; Kechagia et al. 2021; Lamothe et al. 2021; Ren et al. 2020).

To bridge the gap, in this paper, we address the first barrier by defining API misuse identification rules and identifying misclassification cases in the literature, the other two barriers by establishing a curated dataset, proposing a two-dimensional taxonomy, and analyzing real-world API misuses in TensorFlow applications. Ideally, the best practice is to cover as many DL libraries (e.g., PyTorch Paszke et al. 2017, Keras Gulli and Sujit 2017, Caffe Jia et al. 2014, Theano Al-Rfou et al. 2016) as possible. However, the manual labelling process during dataset construction is a highly time-consuming task. Therefore, we finally follow the work of Zhang et al. (2018) to restrict our scope to DL applications of TensorFlow, one representative DL library. Even with such scope, the labelling task in our work spans two months and consumes around 315.2 hours on average for each of the four human labelers. Our dataset could serve as an open-source benchmark, and our associated taxonomy and analysis of API misuses could serve as a reference point for future related research.

The main contributions of our work include:

– **[Fundamentals]** We address a fundamental question around the definition of API misuses in DL applications. We propose two explicit identification rules to help practitioners consensually and unequivocally identify API misuses in DL applications. Based on these

rules, we expose a non-negligible proportion of mislabelled API misuses in two existing bug datasets of DL applications (with respectively 18.2% and 35% mislabelled cases of API misuses).
- **[Dataset]** We build and share the first publicly available dataset for API misuses in DL applications. MISUAPI includes 143 API misuse samples collected from 39 popular TensorFlow projects. We expect MISUAPI to bootstrap several research and engineering efforts around API misuses in DL applications.
- **[Characterisations]** We propose a two-dimensional taxonomy and then conduct an in-depth analysis that investigates the characteristics of API misuses in DL applications in terms of the API-usage element issues, scopes, and symptoms. Among other findings, we highlight that the widely employed MUC taxonomy Amann et al. (2018) fails to cover a third of API misuses that we have encountered and shared in MISUAPI. About 40% API-usage element issues in DL applications do not occur in traditional programs. Among API misuses identified in different code, DL library API misuses significantly differ from general third-party library misuses: the former involves, for a large proportion, issues with wrong parameter types and missing API calls. We also found that, overall, 92.3% of API misuses in DL applications lead to program crashes.
- **[Detectability]** We investigate the possibility of detecting API misuses in DL applications by leveraging four state-of-the-art static analyzers: we observe that 95.8% of API misuses cannot be detected by the existing tools. This finding is a strong call for more research efforts on proposing adapted effective detectors targeting API misuses in DL applications.

The remainder of the paper is structured as follows. We explore the first two research questions in Sections 2 and 3 respectively. To investigate RQ3, we study the characteristics of API misuses in DL applications in Section 4, and employ static analyzers to detect these bugs in Section 5. Then, we draw on practical implications of our study to provide directions for future research in Section 6. In Section 7, we discuss the related work. Finally, we conclude in Section 8.

2 On the Fundamentals of API Misuse

In this Section, we aim to answer the **RQ1** (how can API misuses be defined to ensure that they can be differentiated from other bugs occurring in DL applications?) mentioned in Section 1. The organization of this Section is shown as follows: We first introduce the importance of clear API misuse identification rules in Section 2.1. Then, we recall the definition of API misuse in Section 2.2, and summarize two explicit rules for identifying API misuse in DL applications in Section 2.3. Finally, in Section 2.4, we review the API misuse bugs of DL applications in the literature by using the explicit rules.

2.1 Motivation

API-related bugs are reported to be commonplace in both traditional programs Zhong and Zhendong (2015) and DL applications (Zhang et al. 2018; 2020). One common type of such bugs is "API misuse", which is often studied as an individual and non-trivial bug category in prior work targeting traditional applications (Amann et al. 2016, 2018; Kechagia et al. 2021; Ren et al. 2020; Lamothe et al. 2021; Li et al. 2021; Nielebock et al. 2020; Wen et al. 2019; Zhang et al. 2018; Kechagia et al. 2019). However, in recent studies involving API misuses in DL applications, the API misuses are always mixed with API incompatibility

bugs. Figure 1 presents an example of an API incompatibility problem caused by the updated TensorFlow version. The client code in Keras invokes `softmax()` in TensorFlow 1.3, but it uses TensorFlow 1.5 which has changed the `softmax()` code as the library. This finally led to the error "TypeError: softmax() got an unexpected keyword argument 'axis' while using layers.Dense". An API incompatibility issue is often interpreted as a situation where the invoked API in the project is no longer supported by the third-party library due to the API evolution. Both the case where the name of the API changes and the case where the parameter list changes could lead to API incompatibility issues. API deprecation is an example of such situation, as API deprecation occurs when the invoked API is discarded by the third-party library Scalabrino et al. (2019). Accordingly, such issues are often fixed by updating the version of the third-party library, while API misuse is often fixed by changing the client code. Such API incompatibility issues are categorized into API misuse by Zhang et al. (2020), but they are classified as "API changes" in the study on TensorFlow program bugs Zhang et al. (2018). The inconsistent classification would make practitioners fail to have a clear understanding of API misuse in DL applications.

In the literature, Amann et al. (2016) defined the API misuse as: *"An API misuse is an API usage that violates the API's contract, as opposed to one that does not comply with the client code's logic"*. For example, a contract violation occurs when a method call in the client application passes a null parameter to the API, which requires a non-null parameter and otherwise would raise an exception, without any exception handling actions. Another example is not closing a file stream after finishing writing content into it. In contrast, the bugs where the API usage does not comply with the client code's logic (e.g., querying the wrong database column) are not API misuses. This definition of API misuse has been widely used in the community (Amann et al. 2016, 2018, 2019; Kechagia et al. 2021; Ren et al. 2020; Lamothe et al. 2021; Li et al. 2021; Nielebock et al. 2020; Wen et al. 2019; Zhang et al. 2018; Kechagia et al. 2019). Following this definition, we systematically reviewed the descriptions of API misuses in the studies listed in Table 1, however, we observe that these studies present inconsistent definitions on API misuses. It finally results in the inconsistent classifications on "API misuses", which however could hinder characterizing API misuses in DL applications.

```
x = layers.Dense(classes, activation='softmax')(x) # client code in Keras

def softmax(x, axis=-1):
    # Softmax defined in Keras
    ...
    return tf.nn.softmax(x, axis=axis) # call softmax() in Tensorflow

def softmax(logits, axis=None, name=None, dim=None):
    # Softmax signature in TensorFlow 1.5
    ...

def softmax(logits, dim=-1, name=None):
    # Softmax signature in TensorFlow 1.3
    ...
```

Fig. 1 Code snippet excerpted from the literature Zhang et al. (2020), where the API incompatibility issue is classified into the API misuse category

2.2 Recalling API Misuse Definition

In the definition of API misuse provided by Amann et al. (2016), the API's contract denotes the API usage constraints, which can be obtained by mining the explicit API knowledge from its source code and documentation or extracting implicit API knowledge from its usage examples (Amann et al. 2016, 2018). As presented in Fig. 2, the usage constraint of the API `os.listdir()` is to check if the target directory exists before calling `os.listdir()`. Violating this usage constraint without the non-directory path checking caused the misuse of this API and resulted in the `NotADirectoryError` at runtime.

The definition of API misuse highlights where the usage constraints of the misused API are derived from, i.e., the API providers who develop and expose the API. As discussed by Lamothe et al. (2021) in the systematic review of API evolution literature, the characteristics of API misuse differ from other typical API-related bugs, where the API misuse is categorized into the API usability issues, while other API-related bugs with respect to API migration, API deprecation, and API incompatibility issues are grouped into the API maintenance issues. Our research scope is consistent with this work, i.e., considering API misuse as an individual topic and characterizing it from other API-related bugs.

2.3 Summarizing API Misuse Identification Rules

Based on the review and analysis, we summarize two explicit identification rules for API misuses against other API-related bugs:

1. **Considering sources of usage constraints**. The usage constraints of the misused API originate from the API itself. Therefore, the API source code and documentation are necessary to identify an API misuse.
2. **Excluding relationship with API maintenance.** The basic premise of API misuse identification is that the API version is specified. Therefore, all usage constraints of API misuse come from the individual and specific API version, rather than multiple versions due to API maintenance.

The two rules are consistent with the definition of API misuse Amann et al. (2018) and common practices of API misuse literature review Lamothe et al. (2021) and API misuse dataset construction Amann et al. (2016). The first identification rule identifies the source for obtaining the API usage constraints during misuse identification, and the second rule further helps us distinguish API misuses from other API-related bugs (e.g., incompatibility issues). Such rules further enable us to perform an investigation on the misclassification issues of API misuses in DL applications. The identification rules aim to assist practitioners in collaboratively and unambiguously recognize API misuses in the context of DL applications. To our knowledge, this is the first work to identify the boundary of API misuse in DL applications by presenting a clear definition and actionable rules for identifying API misuses.

```
--- a/facenet/src/facenet.py
+++ b/facenet/src/facenet.py
@@ -568,9 +568,10 @@ def get_dataset(paths):
-   images = os.listdir(facedir)
+   if os.path.isdir(facedir):
+       images = os.listdir(facedir)
```

Fig. 2 A bug fixing commit of an API misuse from the facenet project[3]

Table 2 Results of revisiting API misuse classification in terms of false positives and false negatives

	Zhang et al. (2018)	Humbatova et al. (2020)
# Bugs	175	375
# API bugs	77	20
# API misuses	33	10
# False positives	2	1
# False negatives	12	6

One representative example is that, in prior work such as the work of Islam et al. (2020), we checked all 60 "API bugs" labelled by Islam et al. Based on the API misuse definition and identification rules, we finally identified there exists only one API misuse in these API bugs, of which our check results are publicly available for further check of researchers NEW API (2023). The API misuse definition and identification rules we provided in this paper could ease the process of identifying API misuse from API-related bugs.

2.4 Reviewing API Misuse of DL Applications in the Literature

With the API misuse definition and the two misuse identification rules, we systematically review the literature to investigate to what extent API misuses in DL applications are misclassified. Specifically, we form the search query with boolean "AND" and "OR" operators: *(bug OR fix) AND (deep learning) OR (API misuse)*. We apply the search query on five classic electronic databases (i.e., ACM Digital Library, IEEE Xplore, Springer Link, Elsevier Science Direct, and Wiley) and one search engine (i.e., Google Scholar) to ensure the reliable coverage of state-of-the-art publications. We also consider grey literature Wohlin et al. (2012) (e.g., technical reports, theses) to mitigate publication bias Wohlin et al. (2012). The search was performed on March 28th, 2021.

For each collected publication that involves API misuses in DL applications, we first check if there is any publicly available dataset for our further verification. If such dataset is available, we manually review all API-related bugs and identify API misuses from these bugs, based on the identification rules summarized above. To mitigate the threat of potential subjectivity, four authors independently performed the bug review and misuse identification. Since the manual labelling process involves the independent validation of four authors, we further compute the inter-rater reliability score, i.e., Cohen's kappa coefficient (κ) McHugh (2012) to measure the level of consensus between authors. According to the guideline of Landis and Koch (1977), they characterize $\kappa < 0$ as representing no agreement, $\kappa \in [0.01, 0.20]$ as slight, $\kappa \in [0.21, 0.40]$ as fair, $\kappa \in [0.41, 0.60]$ as moderate, $\kappa \in [0.61, 0.80]$ as substantial and $\kappa \in [0.81, 1.00]$ as almost perfect agreement. The κ score in our labelling process between all the authors ranges from [0.79, 0.95], indicating at least a substantial agreement of the labelling process.

For the inconsistent labels, the authors discussed until an agreement is achieved. All the labels and annotations of the process are made publicly available for researchers or potential users to further validate the results DehengYang (2023).

Table 1 presents the identified literature that involves the study of API misuses in DL applications. By manually checking their statement on API misuse, we observe that almost half of (i.e., Chen et al. 2021, Zhang et al. 2019, Islam et al. 2019b, Wu et al. 2021) these publications give no explicit definition for API misuses in DL applications. Furthermore, for publications that explicitly provide a definition of API misuses, the inconsistency between the

```
--- a/adversarial_crypto/train_eval.py
+++ b/adversarial_crypto/train_eval.py
@@ -128,13 +128,13 @@ def model...
    if key is not None:
-       combined_message = tf.concat(1, [message, key])
+       combined_message = tf.concat([message, key], 1)
```

Fig. 3 An API misuse labelled as an API change bug. The issue report: The arguments for tf.concat were in the opposite order of the API documentation

definitions of publications (e.g., the aforementioned example illustrated in Fig. 1) also exists. These issues pose threats to the stable identification of API misuses in DL applications. On the other hand, among the nine publications, we found that Zhang et al. (2018); Humbatova et al. (2020); Chen et al. (2021), and Islam et al. (2020) provide publicly available datasets. Among the four publications, the API misuses provided by Chen et al. (2021) are related to a specific DL application, i.e., the deployment of DL models on mobile devices, which is limited to analyze the overall characteristics. Islam et al. (2020) provided a large dataset that contains 415 bugs from Stack Overflow and 555 bugs from GitHub, but API misuse is not explicitly identified in the dataset. To be conservative, we exclude the dataset in our review process. Thus, we obtain two datatsets in Zhang et al. (2018); Humbatova et al. (2020). Unfortunately, as reported in the literature Zhang et al. (2018); Humbatova et al. (2020), the two bug datasets only contain 33 and 10 API misuses respectively, which may be insufficient to support a comprehensive analysis, as compared to the popular API misuse dataset containing 90 Java API misuses Amann et al. (2018).

✍ ❝***Finding 1:*** *We observed a lack of explicit definition of API misuse for almost half of reviewed literature as well as the existence of inconsistent definitions, which may threaten the stable identification of API misuses. Furthermore, two datasets with explicit API misuses contain no more than 33 API misuses in DL applications, which further motivates our dataset construction and in-depth analyses.* ❞

To verify if there exist any misclassification cases of API misuses in DL applications, we perform the bug review and API misuse identification on the two datasets. Specifically, we first select all bugs that are labelled as API-related bugs by the dataset constructors. This leads to a total of 77 API-related bugs (i.e., 44 API change bugs and 33 API misuse bugs Zhang et al. 2018) and 20 API-related bugs (i.e., ten API misuses and ten other API-related bugs Humbatova et al. 2020). Then, for each API-related bug, we identify two classes of API misuse misclassification: 1) **false positive:** the non-API misuse was mislabelled as an API misuse, 2) **false negative:** the API misuse was not identified as an API misuse.

Table 2 shows the number of false positives and false negatives of API misuse identification in two datasets. We observed a total of 14 (18.2% = 14/77) mislabelled cases in the first dataset, including two false positives and 12 false negatives. For the second dataset, we observed one false positive and six false negatives, accounting for 35% (=7/20) in total. A wrongly-labelled example is shown in Fig. 3. The call of `tf.concat` in the buggy code passed incorrect parameters (i.e., with the reverse order), which violates the parameter type constraints of the API, resulting in "TypeError" finally[4]. The content of the footnote demonstrates that the bug is an API change bug. API change bugs occur when the behavior of an API changes unexpectedly, often due to updates or changes to the implementation of API. This can result

[4] https://github.com/tensorflow/models/pull/1532

in code that previously worked correctly no longer functioning as expected, leading to bugs or other issues.

✍ ❝***Finding 2:*** *The systematic review of the bug datasets of DL applications reveals the insufficient number of API misuses and a non-negligible proportion of misclassification cases, which reveals the need to curate a dataset exclusively for API misuses in DL applications* ❞

3 Dataset Curating

In this Section, we target at the **RQ2** (where can we collect a significant number of API misuse samples from real-world DL applications to publicly release for the community?).

3.1 Motivation

A large-scale high-quality dataset could significantly boost the development of the associated research area (e.g., the widely used Defects4J dataset Just et al. 2014 for automated program repair). Constructing a large-scale dataset of real-world API misuses in DL applications is important in helping increase the reliability of taxonomy construction and providing an infrastructure for future API misuse detection or repair tool evaluation. Furthermore, a larger dataset with a focus on API misuse in DL applications could help researchers understand the common patterns and issues that arise in different DL applications. While existing datasets are still limited to a small sample size, we propose to build a dataset exclusively for API misuses in DL applications and the characterizing task.

3.2 Overview

Figure 4 describes our semi-automated approach to constructing the dataset. We first collect the commits with respect to bug fixes with related keywords from real-world open-source projects, and filter out non-bug fixes by leveraging the abstract syntax tree (AST) differencing tool GumTree Falleri et al. (2014). To avoid bias in the automated identification and validation, we manually identify the commits related to fixing API misuses from a sampled dataset. The details of our approach are presented as follows.

3.3 Selecting Subjects

There are over 10 deep learning frameworks (e.g., TensorFlow Abadi et al. 2016, Caffe Jia et al. 2014, Keras Gulli and Sujit 2017, PyTorch Paszke et al. 2017) released for various deep learning tasks. In this paper, we follow the workaround of Zhang et al. (2018) to focus on TensorFlow framework, which is the most popular framework in the year 2021 according to the number of TensorFlow-dependent projects (i.e., 104,501) Tensorflow (2021). We employ GitHub search API Github API (2021) to collect TensorFlow projects in GitHub. First, we form a query "tensorflow" to search all TensorFlow-related projects. Then, we fed a query "import tensorflow" into GitHub search API to filter out these projects that are TensorFlow-related but have no source code using TensorFlow APIs. With such strategy, we finally collected 200 top-rated TensorFlow dependents in terms of their starred times (on April

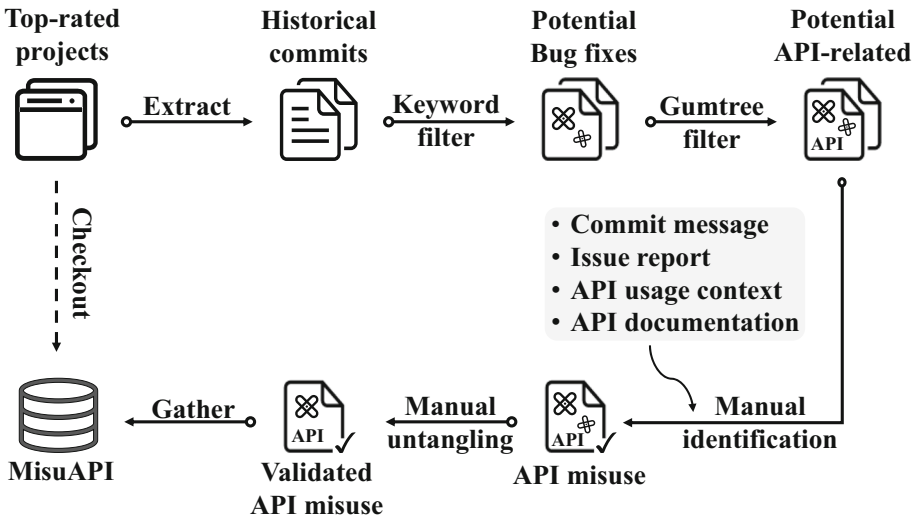

Fig. 4 Overview of the dataset curating approach

1st, 2021), of which the largest starred times reach up to 69,267 and the smallest is 1,143. Following the practice of Dilhara et al. (2021), we use the term "top-rated" in this paper to denote the projects with the largest stargazers counts.

3.4 Collecting Commits Related to API Issues

The 200 TensorFlow dependent repositories contain 135,548 history commits. First, we apply a keyword matching strategy (checking whether the commit message contains the related keywords), which is widely adopted in the literature (Zhang et al. 2018; Humbatova et al. 2020; Zhong and Zhendong 2015; Liu et al. 2018), to collect commits related to bug fixes. Specifically, we choose the keywords by combining the keyword list considered in previous DL bug studies (i.e., "bug", "defect", "issue", "problem", "error", "fault", "fail", "wrong", "nan", "inf", "crash", "fix", "solve", and "repair") Zhang et al. (2018); Humbatova et al. (2020) to ensure comprehensive coverage of potential bug fixes. Through this stage, 33,510 commits related to bug fixes are collected. Note that some existing tools (e.g., GitcProc Casalnuovo et al. 2017) can be used for the filtering process, while we adopt Gumtree to parse commits and identify API changes.

"Bug fixing" related keywords would be used to describe changes of non-API related bug fixes. We further employ GumTree[5], an AST differencing tool, to parse the code changes which are used to identify the API-related modifications by checking if there is any modification (i.e., insert, delete, update, or move Falleri et al. (2014)) on the API element (i.e., represented as the `atom_expr` in GumTree output). As our scope is to investigate API misuses in DL applications, changes to any API used in the DL application will be identified as API-related commits. Finally, 19,395 commits are left as potential bug-fixing commits of API-related issues.

[5] https://github.com/GumTreeDiff/gumtree/tree/v3.0.0-beta1

Table 3 Results of curating dataset

Stage	Number
All commits	135,548
Commits filtered with keywords	33,510
Commits filtered with Gumtree	19,395
Sampling commits	3,639
Commits after manual identification	123
API misuse cases after manual untangling	143

3.5 Manual Identification and Untangling

The number of 19,395 commits is still too large for the manual identification and validation activity. To resolve this issue, we randomly select 80 projects without any bias, of which corresponding 3,639 commits are then selected from the 19,395 ones. Even with such a sampling, the sample size reaches the largest manual validation size compared to related studies on bugs in DL applications (Zhang et al. 2018; Humbatova et al. 2020; Islam et al. 2019a). Furthermore, the manual identification and untangling process in this paper spanned two months and consumed around 315.2 hours on average for each of the four authors. Specifically, API misuse *identification* is to identify whether the bug fixing commit includes at least an API misuse. API misuse *untangling* aims to distill the changes of API misuses from other unrelated changes.

In the identification process, we follow the explicit identification rules presented in Section 2 for API misuse identification. Specifically, we carefully read the commit message, linked issues/bug reports, the buggy API usage context, as well as API documentation, to verify whether the API-related buggy code violates the API usage constraints. To ensure the quality of our dataset, four authors of this paper independently labelled each of API misuses for the sampled 3,639 commits, respectively. The κ score in our labelling process between all the labelers ranges from [0.72, 0.93], indicating at least a substantial agreement of the labelling process. For the inconsistent labels, the four authors discuss until an agreement is achieved. Finally, 123 commits are identified as API misuse fixing commits. In the untangling step, we further perform manual efforts to untangle the code changes unrelated to API misuses based on our previous identification results, to obtain a curated dataset. Eventually, we collect 143 bugs of API misuse in DL application to build our dataset, as it is not rare case where one commit contains multiple API misuses Amann et al. (2016). All the results are summarized in Table 3.

Note that we include all DL-specific API misuses and non-DL-specific API misuses that reside in the DL projects. Our fundamental intuition is that all API misuses rather than just DL-specific ones deserve to be studied, as all these misuses could bring runtime crashes to the programs. Concretely, our dataset is collected from DL applications, which not only use the DL-specific APIs, but also heavily rely on general third-party APIs. Similarly, the dataset collected by Zhang et al. (2018) also contains bugs that are unrelated to Tensorflow, i.e., the bugs are general bugs, not DL-specific bugs.

The inner misuses are also part of the API misuse in DL applications. We would like to clarify that the inner API misuse is not specifically related to DL library API. Figure 5 shows the example of inner API misuse in DL applications. In line 101 of the Python file "audio.py", the developer created a function named *_griffin_lim* with two parameters (*S* and *hparams*). But when the developer called the function at line 77, there was only one parameter.

```
--- a/datasets/audio.py
+++ b/datasets/audio.py
@@ -74,7 +74,7 @@ def inv_linear_spectrogram(linear_spectrogram, hparams):
                 y = processor.istft(D).astype(np.float32)
                 return y
         else:
-                return _griffin_lim(S ** hparams.power)  # line 77
+                return _griffin_lim(S ** hparams.power, hparams)

        ...

        def _griffin_lim(S, hparams): # line 101
            '''librosa implementation of Griffin-Lim
            Based on https://github.com/librosa/librosa/issues/434
            '''
            angles = np.exp(2j * np.pi * np.random.rand(*S.shape))
            S_complex = np.abs(S).astype(np.complex)
            y = _istft(S_complex * angles, hparams)
            for i in range(hparams.griffin_lim_iters):
                    angles = np.exp(1j * np.angle(_stft(y, hparams)))
                    y = _istft(S_complex * angles, hparams)
            return y
```

Fig. 5 The example of the inner-project API misuse in a real-world DL application Mama (2023)

This function call is a typical inner-project API misuse, which could be also common in the practical development activities of practitioners.

So far, there has been no comprehensive system evaluation for DL application API misuse because of the lack of a dedicated dataset for such purposes. To this end, we construct and provide the dataset MisuAPI exclusively for API misuses within DL applications. The availability of a curated dataset could always create new momentum for the associated research areas (e.g., Defects4J Just et al. 2014 for general Java bugs, and Mubench Amann et al. 2018 for Java API misuses). To our knowledge, MisuAPI is the first publicly available dataset exclusively for API misuses in DL applications. Furthermore, in terms of the scale of dataset, comparing against the popular API misuse dataset in Java (i.e,, Mubench that consists of 90 API misuses), our MisuAPI contains 53 more API misuses. MisuAPI could be used for an in-depth analysis of API misuses in DL applications as well as an evaluation for future automated API misuse detection or repair techniques.

✿ Publicly Available Dataset MisuAPI

We construct MisuAPI, the first publicly available dataset exclusively for API misuses in DL applications. MisuAPI contains 143 API misuses collected from 39 popular TensorFlow dependent projects and is publicly available at https://github.com/DehengYang/MisuAPI.

4 Characteristics

In this section, we aim to investigate the characteristics of API misuses in DL applications based on the curated dataset. To present a comprehensive view of the characteristics to developers or researchers, we first present a two-dimensional taxonomy of misused APIs

in DL applications. Then, we dissect API misuses in terms of the proposed categories, and describe the corresponding symptoms that are caused by API misuses.

4.1 Motivation

Taxonomy is a scheme of classification Usman et al. (2017) and has been intensively employed by Software Engineering to represent the theory of a specific field and support the exploration of new knowledge related to the field (Humbatova et al. 2020; Amann et al. 2018; Institute of Electrical and Electronics Engineers 1987; Forward and Lethbridge 2008; Unterkalmsteiner et al. 2014; Šmite et al. 2014). Taxonomy plays a critical role in knowledge representation and discovery, especially for the scarcely explored topic (e.g., API misuse in DL applications). As revealed by prior studies (Zhang et al. 2018; Humbatova et al. 2020; Zhang et al. 2020; Chen et al. 2021; Islam et al. 2019a; Zhang et al. 2019; Islam et al. 2020), there exist significant differences between bugs in traditional and DL applications. Therefore, it is worth proposing a new taxonomy exclusively for API misuse in DL applications to facilitate an in-depth study. In this paper, we construct and compare our taxonomy against existing taxonomy to explore the unique categories of API misuse in DL applications. Furthermore, the taxonomy enables us to perform an in-depth dissection and distill new findings of API misuse in DL applications.

4.2 Taxonomy of Misused APIs in DL Applications

Typically, there are four main types of taxonomy Usman et al. (2017): 1) hierarchy; 2) tree; 3) paradigm; 4) faceted analysis. In this work, to systematically investigate the characteristics of API misuses in DL applications, we propose to build a paradigm-based taxonomy. A paradigm-based taxonomy is generally represented as a two-dimensional structure where entities are presented by the intersection of two specified attributes (Usman et al. 2017; Kwasnik 1999). Specifically, our paradigm-based taxonomy consists of two dimensions of interest: ***API-usage element*** Amann et al. (2018) and ***Scope***. ***API-usage element*** is a program element that appears in API usages Amann et al. (2018). This dimension includes five types: *Incorrect parameter*, *Missing API call*, *Missing checking condition*, *Missing exception handling* and *Incorrect API call sequence*. ***Scope*** is related to the sources of the misused API, where the API knowledge is defined (e.g., the TensorFlow library that provides the misused API `tf.concat()` in Fig. 3). Specifically, the ***Scope*** refers to "scope" as the scope in which the API call is defined, as opposed to the scope in which it is misused. It includes three categories: *DL library*, *General third-party library* and *Inner-project*. To obtain a comprehensive understanding of API misuses in DL applications, we consider API misuses that derive from the project itself (i.e., inner-project misuse) and third-party libraries.

The inner-project misuse and third-party libraries in DL applications are also indispensable topics since the DL applications not only use the DL-specific API, but also use other types of API (e.g., the API related to data loading). The analysis of other types of API misuse in DL applications could help researchers be aware of the existence of general API misuse. The ***Scope*** could pinpoint the sources where the API usage constraints reside for providing insights into future API misuse detection techniques. Similarly, we find that the literature Zhang et al. (2018) also claims some bugs in TensorFlow programs are unrelated to TensorFlow. Thus, it is still necessary to discuss all the API misuse in DL applications, not limited to DL-specific API misuse.

```
--- a/tacotron/models/tacotron.py
+++ b/tacotron/models/tacotron.py
@@ -377,7 +377,7 @@ def add_optimizer(self, global_step):
    optimizer = tf.train.AdamOptimizer(...)
    gradients = optimizer.compute_gradients(
        self.tower_loss[i],
-       list_var=update_vars)
+       var_list=update_vars)
```

Fig. 6 Example of a misused API with incorrect parameter type Mama (2021)

We select the ***API-usage element*** issue as the first dimension of our taxonomy as it could be used to indicate the bug patterns of API misuses as well as the corresponding fixing strategies. In terms of ***scope***, we include it as the second dimension as it pinpoints the sources where the API usage constraints reside for providing insights to future API misuse detection techniques. This taxonomy serves as a basis for our exploration of the characteristics of API misuses in DL applications.

Incorrect Parameter Type means the API is misused by passing a parameter with incorrect type(s). Figure 6 shows the API `compute_gradients()` defined in TensorFlow is misused with an incorrect parameter that finally raises a `TypeError` at runtime. **Incorrect Parameter Value** means the API is misused by passing a parameter with incorrect value(s). Figure 7 shows the value of the `dtype` parameter in `tf.ones_like()` is expected to be `tf.int32` as `tf.scatter_nd()` does not support `bool` on GPUs. **Missing API Call** denotes the absence of an API call required by the API usage constraints, such as the example shown in Fig. 8, where the `tf.device()` required by the `dequeue()` is not called as expected in the client project. **Missing Condition** misused API is caused by the lack of a required condition checking. Figure 9 shows a misused API `shape2d()` that will lead to a RuntimeError since it cannot operate on a parameter with `None` type without the null check of the parameter `resize`. **Missing Exception Handling** means the project does not take actions to handle exceptions that may be raised by the used API. As shown in Fig. 10, the absence of an exception handling for the API `run()` from TensorFlow library may result in a RuntimeError. **Incorrect API call sequence** represents the incorrect order of called APIs that violates the API usage constraints. In Fig. 11, the API `join()` in `multiprocessing.Pool` module explicitly claims that `close()` or `terminate()` should be called before `join()`. The incorrect API call sequence of `join()` in Fig. 11 results in a memory leak.

Misuse of DL library API: this refers to the misused API that is from DL libraries/frameworks, such as TensorFlow. The programming diagrams of DL applications (i.e., data-driven)

```
--- a/tf_quant_finance/math/segment_ops.py
+++ b/tf_quant_finance/math/segment_ops.py
@@ -109,10 +109,13 @@ def segment_diff(x,
    needs_fix = tf.scatter_nd(
        fix_indices,
-       tf.reshape(tf.ones_like(fix_indices, dtype=tf.bool), [-1]),
+       tf.reshape(tf.ones_like(fix_indices, dtype=tf.int32), [-1]),
```

Fig. 7 Example of a misused API with incorrect parameter value tf-quant-finance (2021)

```
--- a/slim/train_image_classifier.py
+++ b/slim/train_image_classifier.py
@@ -450,7 +450,8 @@ def main(_):
-   images, labels = batch_queue.dequeue()
+   with tf.device(deploy_config.inputs_device()):
+     images, labels = batch_queue.dequeue()
```

Fig. 8 Example of a misused API with missing API call tensorflow (2021)

```
--- a/tensorpack/dataflow/image.py
+++ b/tensorpack/dataflow/image.py
@@ -27,7 +27,9 @@ def __init__(...
    channel == 1 else cv2.IMREAD_COLOR
-   self.resize = shape2d(resize)
+   if resize is not None:
+       resize = shape2d(resize)
+   self.resize = resize
```

Fig. 9 Example of a misused API with missing condition tensorpack (2021a)

```
--- a/tensorpack/train/trainer.py
+++ b/tensorpack/train/trainer.py
@@ -91,7 +91,10 @@ def run(self):
-   self.sess.run(self.close_op)
+   try:
+       self.sess.run(self.close_op)
+   except RuntimeError:    # session already closed
+       pass
```

Fig. 10 Example of a missing API with missing exception handling tensorpack (2021b)

```
--- a/spleeter/commands/separate.py
+++ b/spleeter/commands/separate.py
@@ -147,8 +147,8 @@ def process_audio(
    # Wait for everything to be written
-   pool.join()
    pool.close()
+   pool.join()
```

Fig. 11 Example of a misused API with incorrect API call sequence spleeter (2021)

are significantly different from that of traditional applications (i.e., logic-driven) Zhang et al. (2018, 2019), the DL library APIs emerge as a new category of APIs comparing with the traditional programs. The API `compute_gradients()` shown in Fig. 6 is a DL library API defined in TensorFlow. **Misuse of general third-party API:** this denotes the misused API that is defined in the general third-party libraries but not DL frameworks. For example, the misused API `join()` displayed in Fig. 11 is an API defined in the `multiprocessing.Pool` module, a general third-party library used by spleeter[6]. **Misuse of inner-project API:** the misused inner-project API denotes that the API is defined inside the DL application itself. The example shown in Fig. 9 is a misused inner-project API defined in the DL application (i.e., tensorpack[7]).

In the community of API misuses, MUC Amann et al. (2018) is a two-dimensional taxonomy of API misuses derived from a total of 100 Java API misuse examples, including 14 categories with two dimensions: 7 types of API-usage elements (e.g., method call, condition) and 2 kinds of violation types (i.e., missing and redundant), which has been widely adopted in API misuse studies on traditional programs (Kechagia et al. 2021; Li et al. 2021; Nielebock et al. 2020; Kechagia et al. 2019; Bonifacio et al. 2021). However, when applying the traditional taxonomy MUC to the API misuse in DL applications, we note that *Incorrect parameter type*, *Incorrect parameter value*, and *Incorrect API call sequence* cannot be covered by the categories of MUC, which occupy up to 40% (=(30+24+3)/143) of API misuses in our curated dataset MISUAPI. In other words, *missing API call*, *missing condition*, and *missing exception handling* were already discovered by the traditional taxonomy MUC, and there are works (Ren et al. 2020; Zhang et al. 2018) that analyze *Incorrect API call sequence* of traditional software. Thus, our taxonomy includes two new types (i.e., *Incorrect parameter type* and *Incorrect parameter value*).

To verify if the *Incorrect parameter type* and *Incorrect parameter value* misuses are unique, we further check studies of API misuses in other applications. Since there is still no available study of API misuses in general Python applications, to our knowledge, the comparison with such applications is not supported yet. On the other hand, we check a recent study of API misuses in C programs Gu et al. (2019). Although Gu et al. (2019) do not propose a taxonomy, they present a detailed analysis of the root causes of API misuses. One closely related root cause is improper parameter usage, but it mainly indicates the missing preconditions when calling an API. Another closely related root cause is improper causal function calling that denotes a redundant or missing calling of an API. The *Incorrect parameter type*, *Incorrect parameter value*, and *Incorrect API call sequence* are also not covered by the study of API misuses in C programs.

Note that *Incorrect parameter type* and *Incorrect parameter value* category may not be specific to deep learning applications but may be more related to dynamically typed languages (e.g., Python). In dynamically typed languages, the types of parameters are not explicitly declared, and this can lead to errors related to incorrect parameter usage.

> ✍ **"*Finding 3:* *The misused APIs in DL applications can be summarized into Scope and API-usage Element issue categories, of which 40% misused APIs cannot be covered by the widely used API misuse taxonomy for traditional programs.* "**

Compared to existing studies listed in Table 1, our work is the first to present a taxonomy for systematically understanding API misuses in DL applications, while those studies mainly focus on a larger scope that covers various types of bugs in DL applications including API

[6] https://github.com/deezer/spleeter

[7] https://github.com/tensorpack/tensorpack

misuses. Table 4 presents the statistics of the API misuses in MISUAPI in terms of the two dimensions of our taxonomy. Based on this Table, we further dissect the two dimensions in the following sections.

4.3 Dissecting API Misuses with API-usage Element Issues

We first investigate the characteristics of API misuses in DL applications by dissecting them with the API-usage element issue category The "Total (ratio)" column of Table 4 overviews the distribution of each API-usage element issue of API misuses in DL applications. *Missing condition* is the most common API-usage element issue, accounting for 32.9% (=47/143) of all misuses. The second most common API-usage element issue is the *incorrect parameter type* with 30 misused cases in MISUAPI. It is followed with the incorrect parameter value, *missing API call*, and *exception handling* that account for 24, 21, and 18 cases, respectively. The *incorrect API call sequence* as the least common API-usage element issue only includes three cases.

Compared with traditional applications of Java API misuses Amann et al. (2018), DL applications also suffer from API misuses about *missing API call* (12.6% vs. 27.3%), *missing condition* (32.9% vs. 43.6%) and *missing exception handling* (14.7% vs. 9.1%). Such commonality is, however, ignored by prior studies (as listed in Table 1) involving API misuses in DL applications. As pointed out by Dilhara et al. (2021), DL applications also share some similarities with traditional programs. Thus, it is not surprising to observe the common API-usage element issues in DL applications. Instead, such commonality could serve as hints for future proposals of automated detection or repair approaches toward these issues in DL applications.

[Distinctive API Misuses] Because of the specification of Python code and the DL neural network models, *Incorrect parameter type*,*Incorrect parameter value*, and *Incorrect API Call Sequence* are the three kinds of distinctive API misuses for DL applications, which significantly differ from that of traditional programs (e.g., Java programs Amann et al. 2018) where incorrect parameter type/value misuse and incorrect API call sequence has not been observed in past work.

Table 4 Statistics of the API misuses in MISUAPI

	DL library API	General third-party library API	Inner Project API	Total (ratio)
Incorrect parameter type	15	8	7	30 (21.0%)
Incorrect parameter value	6	15	3	24 (16.8%)
Missing condition	8	35	4	47 (32.9%)
Missing exception handling	2	17	2	21 (14.7%)
Missing API call	16	0	2	18 (12.6%)
Incorrect API call sequence	2	1	0	3 (2.1%)
Total (ratio)	49 (34.3%)	76 (53.1%)	18 (12.6%)	143 (100%)

The misuse of *incorrect parameter value* means that the parameter is of the correct type, but has an incorrect value. The usage constraints for this subcategory are more related to the internal code logic of the API, thus are always implicitly encoded in the API source code. In the dataset MISUAPI, the *incorrect parameter value* results in 24 API misuses with incorrect parameter, as shown in Table 4.

The misuse of *incorrect parameter type* denotes that the parameter passed to the misused API is of incorrect type. The usage constraints for this subcategory are often explicitly encoded in the API source code or documentation. Even though, there are still 30 API misuse cases caused by the *incorrect parameter type*, as shown in Table 4. One straightforward reason lies in Python programming language used by DL applications, as Python executes type checking only at runtime, which fails to ensure the legality of the parameter type passed to the API before execution and finally leads to the runtime error.

✍ **"Finding 4:** *The distinctive incorrect parameter type and value take a dominant proportion (i.e., (30+24)/143=37.8%) of API-usage element issues of API misuses in DL applications.* **"**

4.4 Dissecting API Misuses with Scope

In this section, we investigate where the misused APIs derive from according to the scope category defined for API misuses in Section 4.2. The "Total (ratio)" row of Table 4 presents the distribution of API misuses in terms of the scope category. Most misused APIs are derived from *general third-party libraries* that occupy 53.1% (=76/143) of all misused APIs. As the specific scope of misused APIs, *DL library* presents a non-trivial percentage (34.3% = 49/143) for API misuses in DL applications. It also reminds that, DL application developers should also pay attention to the invocation of APIs defined their inner-projects, which could lead to API misuses.

✍ **"*Finding 5:*** *General third-party library (53.1%) is the primary source of misused APIs in DL applications. And it is notable that one out of three misused APIs in DL applications is from the referenced DL libraries.* **"**

Differences among the Three Scope Categories To analyze the potential relationship among the three scope categories of API misuses, we first measure the commonality across these categories by calculating Spearman correlation coefficient (i.e., Spearman's ρ Zar, 2005). Spearman correlation coefficient is a statistical test used to indicate the direction (positive or negative) of a relationship between two paired variables. Such coefficient is also employed by prior studies involving bugs in deep learning-related applications Shen et al. (2021). Specifically, we rigorously follow the guideline Zar (2005) to calculate the Spearman's ρ based on the rankings of the API-usage element issues for each scope category.

For the value of Spearman's $\rho \in [-1, 1]$, the higher absolute value (i.e., $|\rho|$) represents a higher degree of commonality between the two scopes, otherwise, a lower degree of commonality.

The correlation among three scope categories is presented in Fig. 12. The DL library API misuses show a weak correlation (i.e., 0.15) with the general third-party library API misuses, which indicates a low commonality on the API-usage element of API misuses between the two categories. Nevertheless, the inner-project API misuses are a relatively high correlative with that of DL library API misuses (i.e., 0.76) and general third-party library API misuses

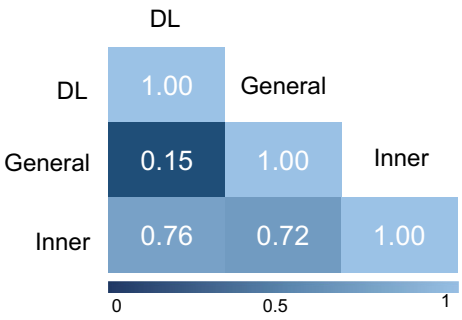

Fig. 12 Correlation among each pair of scopes, as indicated by Spearman's ρ score. DL, General and Inner represent "DL library API", "General third-party library API" and "Inner-project API", respectively

(i.e., 0.72). It might result from the common truth that the inner-project code is a kind of bridge connecting the DL libraries and general third-party libraries to develop the DL application.

Looking at the number of misused APIs caused by *incorrect parameter type* and *incorrect parameter value*, The *DL Library API* category includes 15 incorrect type cases and 6 incorrect value cases. In contrast, the *General Third-party library API* category includes 8 incorrect type cases and 15 incorrect value cases. When comparing with the usages of *General Third-party library APIs*[8], DL library APIs often operate on parameters with more complex data structures (e.g., Tensors or Arrays) that are used for building learning models. Therefore, the misused parameter types in DL library API misuses are higher than the misused parameter values. Additionally, as stated by Dilhara et al. (2021), DL libraries are more frequently updated than general traditional ones, which leads to the fast evolution of APIs and their documentation in DL libraries. This could eventually result in developers cannot fully catch up with the changes related to the parameters of DL library APIs.

For the *Missing API call* caused API misuses, most of them (i.e., 16/18=89%) are from the *DL library API*, but none of such misused API is from the *General third-party libraries*. Although APIs from DL libraries and general third-party libraries are used for deep learning tasks, the DL library APIs always play a pivotal role in constructing the learning model with complex network structure, where the API call sequences are always used. So, the *missing API call* caused API misuses mainly drive from the DL libraries, and the called DL library APIs are often dependent on each other in the network. The missing API call could arise due to developers' misunderstanding/mistake when deploying these APIs. For example, training the deep learning model always has a dependence on specific devices (e.g., CPUs or GPUs). As shown in Fig. 13, the developer of the `prettytensor` application missed the call of `tf.device('/cpu:0')`: mandated by the `tf.nn.embedding_lookup()`[9] that finally led to a crash.

On the contrary of *Missing API call*, *General third-party library API* category contains more API misuses of *missing condition* (i.e., 35, as shown in Table 4) and *missing exception handling* (i.e., 17, as shown in Table 4) than the *DL library API* category. APIs from general third-party libraries mainly undertake data pre-processing tasks (e.g., loading data from disks with `json.load()`) and some basic logic operating tasks that are always connected to some specific conditions or exceptions. Table 5 presents 30 third-party libraries that are the API providers of API misuses with respect to the 35 *missing condition* cases and 17 *missing exception handling* cases. Specifically, the majority of third-party libraries

[8] The *Inner-Project API* category contains seven incorrect type cases and three incorrect value cases, which is consistent with *DL Library API* category, since the development of inner-project APIs of DL applications is closer to the invocations of *DL library APIs*.

[9] https://github.com/google/prettytensor/commit/01ee67d6e0cc5e9d6ae5f07045024a638564fe78

```
--- a/prettytensor/tutorial/baby_names.py
+++ b/prettytensor/tutorial/baby_names.py
@@ -52,7 +52,9 @@ def create_model(text_in,
-    embedded = text_in.embedding_lookup(CHARS, [EMBEDDING_SIZE])
+    # The embedding lookup must be placed on a cpu.
+    with tf.device('/cpu:0'):
+      embedded = text_in.embedding_lookup(CHARS, [EMBEDDING_SIZE])
```

Fig. 13 Example of a misused DL library API that depends on specific devices (i.e., CPU) prettytensor (2021)

(i.e., 19/32=59.4%) involve in the data processing locally (e.g., `json`) or globally (e.g., `requests`). Note that such data processing often requires interactions with disks or internet, various logic and exceptions should be carefully addressed. Thus, the missing condition and exception handling caused API misuses are mainly from the general third-party libraries. By listing the categories and libraries in Table 5, it might potentially remind researchers or developers to pay attention to these libraries when the researcher is designing a new approach or when developer is programming.

✍ **"*Finding 6:* *API misuses from the DL library APIs present a totally different distribution from the General third-party library APIs. As a bridge between the DL library APIs and the General third-party library APIs, the API misuses from inner-project APIs present a relatively high correlation to both of them. In addition, because of the differences between the concrete developing tasks targeted by the DL library APIs and General third-party library APIs, API misuses from the DL libraries are mainly caused by incorrect parameter types and missing API call, but API misuses from the General third-party libraries are mainly caused by the incorrect parameter values, missing condition, and missing exception handling.*"

4.5 Symptoms

In this section, we aim to investigate the symptoms caused by API misuses in DL applications. To that end, we manually check the following sources: 1) code changes within the commit, 2) commit message, 3) issue report, and 4) API documentation, to summarize the categories of symptoms. We first collect clues from the bug fixing commit of fixing API misuses to verify whether there is any code change or description that can provide the hints of the API misuse for summarizing symptoms. We then check the associated issue report that is available in the GitHub repository. Finally, we study the documentation of the misused APIs to confirm the

Table 5 General third-party libraries that correspond to missing conditions and missing exception handling misuses

Categories	General third-party libraries
Basic operations (4)	collections, dict, getattr, string
Data processing (19)	bentoml, cv2, docker, ffmpeg, fig, google, http, io, json, librosa, os, open, pickle, pycocotools, requests, shutil, wave, zmq, msgpack
Others (7)	numpy, pandas, pycuda, random, scipy, threading, tqdm

Table 6 Symptoms for each category of API misuses

	DL library API	General third-party library API	Inner-project API
Error	39	75	18
Memory leak	3	1	0
Race condition	3	0	0
Low efficiency	4	0	0

summarized symptoms. Same to the identification of API misuses, summarizing symptoms is also performed by four authors independently, where the disagreement is eliminated via the group discussion. For the results of independent identification, the κ score ranges from [0.81, 0.96], indicating the almost perfect agreement of the labelling process. Eventually, we summarize four categories of symptoms as follows:

- **Symptom 1: Error.** Error means that the application terminates unexpectedly at runtime. The error could directly result in a program crash, which often provides an error message to indicate the exact information of the error. For example, the misused API in Fig. 9 raises RuntimeError with an error message "Illegal shape: None".
- **Symptom 2: Memory Leak.** Memory leak is a typical vulnerability resulting from the incorrect management of memory locations. The memory leak could lead to low performance of the application or even crashes. One example in Fig. 11 is that the incorrect API call sequence of `close()` and `join()` leads to a memory leak.
- **Symptom 3: Race Condition.** Race condition often occurs when multiple processes or threads depend on some shared state. Apart from program crashes, the race condition could also result in serious security problems, e.g., privilege escalation. An example resulting in this symptom is missing the API call for initializing TensorFlow variables[10].
- **Symptom 4: Low Efficiency.** This symptom indicates that the time cost spent on executing the application is much longer than the expectation of the developer or user. For example, in Fig. 8 the missing call of `tf.device()` makes the application much slower.

As presented in Table 6, Error is the most common symptom for all API misuses in DL applications, accounting for 92.3% (=132/143) of all symptoms. This indicates that the majority of API misuses can result in program crashes. The rest 11 cases all fall into the DL library API misuse, except for one memory leak caused by a general third-party API misuse (i.e., the misuse in Fig. 11). There are six DL API misuses resulting in vulnerability-related issues, including three memory leak cases and three race condition cases. Additionally, DL library API misuses result in low efficiency issues that are not observed in symptoms of other types of misuses. Figure 8 shows an example of a misused DL library API from `tf.contrib.slim` module. If the `dequeue` node is not assigned to `input_device`, the "Ignoring device specification /device:GPU:X for node 'clone_X/fifo_queue_Dequeue'" message will be printed when running the program[11], and the increased number of GPUs even makes the program slower[12].

[10] https://github.com/horovod/horovod/commit/9420ef71c197b544f122a08ccb8db5491afa3548

[11] https://github.com/tensorflow/models/pull/1480

[12] https://github.com/tensorflow/models/issues/1390

Table 7 Information of the four selected static analyzers

Name	Version	Stars	Organization
mypy (2021)	v0.812	10.5k	Python
pyright (2021)	v1.1.130	6.6k	Microsoft
pyre-check (2021)	v0.9.0	5.3k	Facebook
pylint (2021)	v2.7.4	3.3k	PyCQA

✍ **"*Finding 7:* Over 90% API misuses in DL applications will lead to program crashes, and several API misuses could even result in vulnerable issues. It strongly calls for practitioners' attention to addressing API misuses in DL applications. "**

5 Being Detectable

As presented in Section 4.5, API misuses could cause severe issues for DL applications. As pointed out by Zhang et al. (2020), static program analysis tools, which often serve as a bug finder, are promising tools to handle API misuses in DL applications. We thus investigate to what extent API misuses in DL applications can be detected by state-of-the-art static analyzers. Ideally, API misuse detection tools exclusively for API misuses in DL applications should be used for evaluating whether these API misuses could be successfully detected. We first searched for API misuse detectors that could be used in DL applications. We observed that, although API misuse detection techniques have emerged in recent years with promising results achieved on real-world programs (Nielebock et al. 2020; Amann et al. 2016; Wen et al. 2019; Gulli and Sujit 2017), all of these detectors focus on Java API misuses. There is no any API misuse detector targeting Python programs available from the community. Furthermore, it could be a big challenge to apply the API misuse detector targeting Java to Python programs due to syntactic incompatibilities and lack of analysis of dynamic features. Therefore, we resort to state-of-the-art static analyzers for Python programs.

Following the common practice in selecting bug detection tools for specific type of bugs (Kechagia et al. 2021; Eghbali and Pradel 2020), we select four state-of-the-art static analyzers (i.e., mypy 2021, pyright 2021, pyre-check 2021, pylint 2021) targeting python programs from a list of 29 static analyzers SAST (2021), according to the stargazers count of each analyzer in GitHub. Table 7 shows the basic information of the four selected static analyzers that are well-maintained by well-known organizations or big companies such as Microsoft and Facebook.

Specifically, the pipeline of these static analyzers can be summarized as follows:

1. Parsing: The tool parses the Python code into an abstract syntax tree (AST) that represents the structure of the code.
2. Analysis: The tool analyzes the code to identify potential issues and violations of coding standards. This may include type checking, metrics analysis, syntax checking, and other checks depending on the tool.
3. Reporting: The tool generates a report that lists any errors or warnings it found in the code. The report may include details about the location of the issue in the code, a description of the issue, and suggestions for how to fix the issue.

The exact details of the pipeline will vary depending on the tool and the specific configuration used.

Table 8 The exact syntax issue of API misuses detected by static analyzers

	mypy	pylint	pyre-check	pyright
Incorrect parameter type	0	2	0	2
Incorrect parameter value	0	0	0	4

In our experiment, we configure and run the four analyzers based on their official documentation. Specifically, the input of the analyzers is the Python source file that is to be analyzed, and the output is the report of a specific analyzer. The report records the possible issue detected by the analyzer. We then manually check the output of these analyzers to verify if the API misuses can be detected.

Table 8 presents the detected results of the four static analyzers. The numbers in Table 8 means the number of the exact API-usage element issue of API misuse that a static analyzer could detect. Recall that the incorrect parameter type and value account for 37.8% (=54/143) of all 143 API misuses. Only two analyzers (i.e., pylint and pyright) reported the six true positives, while the others failed to detect any API misuses and provided many unrelated warning messages. The six detected API misuses include two incorrect parameter types of inner-project API misuses (i.e., pylint and pyright detected the same two cases) and four incorrect parameter values of general third-party library API misuses. Specifically, all four detected incorrect parameter value misuses are related to the `pickle.load()` API that is from the Python standard library Python (2021), one of them is shown in Fig. 14. Specifically, in Fig. 14, the buggy version has the statement *classifier = pickle.load(open(path))*, which will pass the incorrect value "r" to *open()* API and return a "file" object that *pickle.load()* could not deal with. The main reason is *pickle.load()* must require an "IO[bytes]" parameter instead of "file" object. Thus, the fix contains a statement *classifier = pickle.load(open(path, 'rb'))* that will pass the correct value "rb" to *open()* API, and the patch satisfies the requirements of *pickle.load()*.

The detection results indicate a limited ability of state-of-the-art static analyzers to detect API misuses (i.e., 137/143=95.8% misused APIs cannot be detected by them) in DL applications. In particular, for API misuses from DL libraries, none of them is detected. This is because the analyzers at present are only able to parse the usage constraints related to parameters of a small fraction of APIs defined inside the project or provided by the Python standard library. They fail to mine more complicated usage constraints related to the rest four API-usage element issues.

Furthermore, the automated repair of API misuse in DL applications is the next step of the research pipeline, which requires domain-specific knowledge. We observe a notable absence of efforts specifically directed towards identifying and rectifying API misuse in DL applications. We are confident that the open-source dataset we have created and the analysis

```
--- a/autokeras/classifier.py
+++ b/autokeras/classifier.py
@@ -7,11 +7,12 @@
-   classifier = pickle.load(open(path))
+   classifier = pickle.load(open(path, 'rb'))
```

Fig. 14 The misuse of pickle.load API detected by pyright. The report of Pyright: Argument of type "TextIOWrapper" cannot be assigned to parameter "file" of type "IO[bytes]"

we have undertaken for API misuse in DL applications could serve as valuable resources to support future research in this area.

✍ **“*Finding 8:* Lacking specific API misuse detecting tools for DL applications, over 90% API misuses cannot be detected by the state-of-the-art static analyzers of Python programs. In particular, none of the misused APIs from the DL libraries can be detected. ”**

6 Discussion

6.1 Implications

We discuss the implications of our study for future research directions related to API misuses in DL applications.

With **Findings 1-2**, we highlighted that the literature not only lacks consistent classification criteria for API misuses in DL applications, but also faces data availability issues to support extensive studies related to API misuses in DL applications. We have now addressed these challenges by providing clear identification rules for API misuses in DL applications, as well as a curated dataset. Our rules and dataset are expected to spark renewed interest and research activity in the area of API misuse in DL applications by providing a systematic and robust analysis. By utilizing our rules and real-world dataset, researchers can build upon our work to develop more effective and efficient techniques for addressing the challenges of API misuse in DL applications.

Findings 3-4 highlight three new API-usage element-related API misuses in DL applications that are not observed in traditional applications. *Incorrect parameter type*, *Incorrect parameter value*, and *incorrect API call sequences* account for 39.9% of API misuses in the collected samples. Future automated localization and repair approaches, by focusing on these cases, could alleviate a large proportion of developer burden.

Findings 5-6 suggest that DL libraries and general-purpose libraries should be dealt with differently. Indeed, the former provides functionalities related to DL specific tasks (e.g., initialization and parameterization of complex network models) while the latter serves for more trivial functionalities in data pre-processing, metric computation, common operations, etc. Therefore the misuse types are different, requiring specialized detection and repair approaches for each.

Finding 7 insists on the consequences of API misuses in DL applications. By highlighting that API misuses will likely make the programs crash or even introduce vulnerabilities (e.g., memory leaks and race conditions), we suggest that the community consider API misuses as an important category of bugs to address with automated approaches.

Finding 8 shows that over 90% API misuse cannot be detected by the state-of-the-art static analyzers of Python programs. Python is a kind of dynamic programming language, which makes it difficult to analyze statically. Besides, there are two other possible reasons. The first is that the existing analyzers are not advanced enough to handle the inherent features of Python. The second is the domain-specific knowledge of API misuses of DL applications are not included in the analyzers. Therefore, it urgently calls for more research efforts towards more advanced analyzers and addressing API misuses in DL applications.

6.2 Threats to Validity

When labeling the API misuses, four authors of our paper independently labelled each of API misuses. For the inconsistent labels, we discuss until an agreement is achieved. Although the settings are conducted in prior works (Humbatova et al. 2020; Islam et al. 2019a), there may still exist potential efficiency problems. Independent raters with moderator and reconciliation should be further effective and efficient.

To the best of our knowledge, MISUAPI is the largest dataset of API misuses among existing publicly available datasets of API misuses (Zhang et al. 2018; Humbatova et al. 2020; Amann et al. 2016, 2018), be it for traditional (e.g., MuBench with 77 misuses from 33 projects Amann et al. 2016) or DL applications (i.e., the datasets studied in Table 1). However, our dataset of 143 API misuses may not be representative in the following three aspects.

First, MISUAPI may not be representative is that we only consider the code changes in Python file (i.e., the filename ends with '.py'). Some of the DL applications in GitHub may be written in Python notebooks, which is not included in our work. It could be beneficial to include various kinds of source in the future. Furthermore, we check "import tensorflow" statement in our project filtering out process, which does not guarantee the exclusion of tutorial projects. Second, MISUAPI is originated from only GitHub. However, there are numerous alternative data sources available for collecting instances of API misuse, such as StackOverflow. If API misuse from additional sources were included in the dataset, it could lead to changes in the data distribution and potentially result in different characteristics of API misuse. This represents a latent threat to the dataset's integrity. We plan to include it as a prospective avenue for our future work. In order to include more API misuse from different sources, we also analyze the API bugs from the existing datatset collected by Islam et al. (2020). Eventually, we identified only one API misuse from 60 API bugs in the dataset[13], which highlights the significance of MISUAPI. From a statistical perspective, this identified API misuse will not significantly impact the distribution of our dataset, nor will it affect the characteristics of API misuse in DL applications. Thus, we did not merge it into our dataset. Third, the subject selection in Section 3.3 is for selecting the top-rated DL projects and may fail to avoid low-quality projects. We mitigated this threat by manually identifying the API misuse commits as described in Section 3.5.

Due to the time-consuming process of manual identification and untangling, MISUAPI does not contain the latest API misuse bugs. While many popular real-world bug datasets are typically valued by researchers due to their high-quality bugs (i.e., carefully curated bugs that eliminate unrelated code changes as we also did in Section 3.5 Manual Identification and Untangling), the inclusion of the latest bug may not be a major factor impacting the quality of a dataset. For example, one of the most popular datasets for automated program repair (i.e., Defects4J Just et al. 2014), which was constructed in 2014 and has been used as a benchmark by more than 60 newly proposed APR tools, consists of real-world bugs that appeared in as early as 2007. We believe the current version of our dataset, which is also carefully curated as Defects4J ever did, could bring benefits to the community.

Another threat may be introduced through the manual validation of API misuses during the literature review in Section 2 and dataset construction in Section 3. To mitigate this threat, four authors of this paper independently performed the validation and iteratively refined the

[13] The identification results can be found at: https://doi.org/10.5281/zenodo.8302351. The identification results include whether each API bug is an API misuse and the reason for it being or not being an API misuse.

results until a consensus was reached. All labels and associated annotations are made publicly available DehengYang (2023).

The four static analyzers selected for detecting API misuses in MISUAPI may not represent all static analyzers, which also poses a threat to this paper. We notice that more and more researchers have devoted effort to developing automated bug detection and repair techniques (Wardat et al. 2022; Cao et al. 2022; Wardat et al. 2021; Yan et al. 2021; Yu et al. 2021; Usman et al. 2021) for bugs in DL applications, but there is still no such work focusing on detecting and repairing API misuse in DL applications. The open-source dataset we constructed and the analysis we conducted for API misuse in DL applications could serve as a reference point for future related research.

7 Related Work

7.1 Study on Bugs in DL Applications

Recently, a set of empirical studies has emerged in top venues to reveal the bug characteristics and corresponding debugging challenges. Zhang et al. (2018) studied deep learning applications built on top of TensorFlow and collected program bugs related to TensorFlow from StackOverflow QA pages and GitHub projects. Islam et al. (2019a) expanded the studied DL libraries by including *Caffe*, *Keras*, *Theano*, and *Torch*. Humbatova et al. (2020) introduced a large taxonomy of faults in DL systems. Islam et al. (2020) studied the 415 repairs and found that deep neural network bug fix patterns are distinctive compared to traditional bug fix patterns. Zhang et al. (2020) performed the first comprehensive study on program failures of DL jobs. The main difference between these studies and our work is that they focus on the breadth of research by covering all bug categories in DL learning applications while we aim to explore one important category (i.e., API misuse) in depth.

7.2 Study on API Misuses in DL Applications

There have been several studies that partially or completely involve API misuses, as listed in Table 1. These related work allow us to stand on the shoulders of giants while focusing on creating a curated dataset and acquiring new findings for API misuses in DL applications.

Among the studies that partially investigate API misuses, Zhang et al. (2018) studied 175 bugs in TensorFlow applications collected from GitHub and Stack Overflow, where they identified 33 API misuses in total.

Humbatova et al. (2020) present a taxonomy of bugs in DL-specific bugs via a structured interview with 20 researchers and practitioners. Unlike the work of Zhang et al. (2018), they exclude all generic (i.e., non-DL specific) bugs in DL applications before building the taxonomy, and identify API misuse as a leaf category of the whole taxonomy. Vélez et al. (2022) examined code patches and bug reports of `tf.function`, and found that hybridization approach could lead to API misuses. The work of Vélez et al. (2022) is not an exclusive study targeting API misuse. It focuses on covering all types of bugs of `tf.function`, including API misuse, API incompatibility, numerical errors, etc. Compared with our work, it reveals some API misuses but is limited to a single API (i.e., `tf.function`), while our work does not hold such constraint. We present a two-dimensional taxonomy for API misuses in DL applications, and analyze the characteristics of API misuses in terms of the syntax, API-usage element issues, scopes, and symptoms.

Islam et al. (2019b) manually analyzed 3,243 highly-rated Q&A posts from Stack Overflow and reported that API misuses occur across all stages of machine learning pipelines. Similarly, Zhang et al. (2019) manually studied 715 Q&A posts from Stack Overflow, where they characterized API misuse as one of the five main root causes of issues in DL applications. In addition, Islam et al. (2019a) performed a comprehensive study of bugs in DL applications, and they identified that API misuse is one of the ten root causes of bugs in DL applications.

Chen et al. (2021) studied 304 deployment faults from Stack Overflow and GitHub and reported fixing the API misuse as one of 13 fix strategies for different fault symptoms. Zhang et al. (2020) performed a comprehensive study on program failures of deep learning jobs that run on a remote and shared platform (i.e., Philly from Microsoft), and they revealed a large proportion of API misuses (i.e., 138 API misuses out of 668 DL specific bugs).

The major difference between these studies and our work is that our work is inspired by these studies but further proposes new infrastructure and obtains new findings. Specifically, we notice that although API misuses are frequently mentioned in these studies but are seldom investigated in an in-depth manner. Therefore, we construct the first publicly available dataset, namely MISUAPI, exclusively for API misuses in DL applications. Then, we propose a two-dimensional taxonomy to further characterize API misuses in DL applications. Different from the tree-based taxonomy of Humbatova et al. (2020), to obtain a comprehensive understanding of API misuses in DL applications, our taxonomy focuses fully on API misuses rather than all bugs in DL applications. In addition, our characterization analysis performed based on MISUAPI further reveals the symptoms of each category and whether they could be detected by existing static analyzers.

Note that we do not include StackOverflow for the following reasons: (1) Selection bias: The questions and answers on StackOverflow often only reflect the views and experiences of some people. (2) Quality: Because StackOverflow allows anyone to post questions and answers, there are questions and answers of varying quality. This can lead to noisy, erroneous, or ambiguous information in the dataset. (3) Context: The questions and answers on StackOverflow are usually posed for specific situations, but these situational information may be missing from the dataset. This can lead to difficulties in accurately understanding questions and answering them.

Among the studies that completely investigate API misuses, Wan et al. (2021) focused on the API misuses that reside in the machine learning cloud services. They manually analyzed 360 open-source GitHub projects that use Google Cloud or AWS APIs and reported that over 69% of applications suffer from API misuses in their latest version. In addition, Wu et al. (2021) analyzed a specific type of API misuse (i.e., crashing tensor shape faults) in TensorFlow and Keras applications. and they further constructed a dataset of 146 buggy programs with crashing tensor shape faults. As compared to the two studies, our work focuses on a different scope (i.e., API misuses in popular TensorFlow dependent applications) rather than machine learning cloud API misuse or a specific type of API misuses. Thus, this leads to findings and conclusions in the paper that differ significantly from the two studies.

8 Conclusion

We focus on demystifying API misuses in DL applications, whose study faces four challenges related to the classification of API misuses, the lack of dataset in the literature, the limited understanding of API misuses characteristics and support for their detection. We therefore

start by addressing the fundamental but unanswered question about the definition of API misuse as well as how to identify them in DL applications. Then we construct a curated dataset from real-world popular DL projects and share it with the community. Subsequently, building on the dataset, we perform an in-depth analysis to investigate the characteristics of API misuses in DL applications in terms of API-usage element issues, scopes and symptoms. Finally, we measure the detectability of these API misuses with four state-of-the-art static analyzers. Through these analyses, we provide actionable findings that could be further leveraged by both practitioners to avoid the API misuses and researchers to propose automated approaches for these misuses.

In future work, we plan to develop a fully automated API misuse detection and repair tool based on the distilled findings in our work. In addition, we plan to investigate new API misuses in more DL libraries (e.g., PyTorch Paszke et al. 2017, Keras Gulli and Sujit 2017, Caffe Jia et al. 2014, Theano Al-Rfou et al. 2016) to present a more comprehensive view of API misuses in DL libraries.

Acknowledgements This research was partially supported by the National Natural Science Foundation of China (Nos. 62172214, 62272072), the Natural Science Foundation of Jiangsu Province, China (BK20210279), and the Major Key Projectof PCL (No. PCL2021A06).

Data Availability For the sake of Open Science, we make the replication package with source code and the curated dataset MisuAPI publicly available at: https://zenodo.org/record/7684952

Declarations

Conflict of interests/Competing interests The authors have no relevant financial or non-financial interests to disclose.

Ethics approval No ethics approval was required for this paper.

References

A curated list of static analysis (sast) tools for all programming languages. https://github.com/analysis-tools-dev/static-analysis#python. Accessed June 2021

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al (2016) Tensorflow: a system for large-scale machine learning. In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), p 265–283

Al-Rfou R, Alain G, Almahairi A, Angermueller C, Bahdanau D, Ballas N, Bastien F, Bayer J, Belikov A, The Theano Development Team et al (2016)Theano: a python framework for fast computation of mathematical expressions. arXiv:1605.02688

Amann S, Nguyen HA, Nadi S, Nguyen TN, Mezini M (2018) A systematic evaluation of static api-misuse detectors. IEEE Trans Softw Eng 45(12):1170–1188

Amann S, Nadi S, Nguyen HA, Nguyen TN, Mezini M (2016) Mubench: a benchmark for api-misuse detectors. In Proceedings of the 13th international conference on mining software repositories, pp 464–467

Amann S, Nguyen HA, Nadi S, Nguyen TN, Mezini M (2019) Investigating next steps in static api-misuse detection. In 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp 265–275. IEEE

Artifact page of our study (2023). https://github.com/DehengYang/MisuAPI

Bonifacio R, Krüger S, Narasimhan K, Bodden E, Mezini M (2021) Dealing with variability in api misuse specification. arXiv:2105.04950

Cambronero J, Li H, Kim S, Sen K, Chandra S (2019) When deep learning met code search. In Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 964–974

Cao J, Li M, Chen X, Wen M, Tian Y, Wu B, Cheung S-C (2022) Deepfd: automated fault diagnosis and localization for deep learning programs. In: Proceedings of the 44th international conference on software engineering, pp 573–585

Casalnuovo C, Suchak Y, Ray B, Rubio-González C (2017) Gitcproc: a tool for processing and classifying github commits. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis, pp 396–399

CEO Nvidia (2023) Software is eating the world, but AI is going to eat software. T. Simonite

Chen Z, Yao H, Lou Y, Cao Y, Liu Y, Wang H, Liu X (2021) An empirical study on deployment faults of deep learning based mobile applications. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE), pp 674–685. IEEE

Dilhara M, Ketkar A, Dig D (2021) Understanding software-2.0: a study of machine learning library usage and evolution. ACM Trans Soft Eng Methodol (TOSEM) 30(4):1–42

Eghbali A, Pradel M (2020) No strings attached: an empirical study of string-related software bugs. In: 2020 35th IEEE/ACM international conference on automated software engineering (ASE), pp 956–967. IEEE

Example of a missing api with missing exception handling. https://github.com/tensorpack/tensorpack/commit/132dcccd34a831a01e4fcdbd32f869b36f04537e. Accessed June 2021

Example of a misused api with incorrect api call sequence. https://github.com/deezer/spleeter/commit/55723cfa6296388ea1f584e2591f1d89e4c0afb6. Accessed June 2021

Example of a misused api with missing api call. https://github.com/tensorflow/models/commit/001a260214ba34f36e149bbd24f7f5d6a6634500. Accessed June 2021

Example of a misused api with missing condition. https://github.com/tensorpack/tensorpack/commit/ae84b52ad5402ab1716e0f1e9790ce1da9d706d1. Accessed June 2021

Example of a misused dl library api depending on the specific device. https://github.com/google/prettytensor/commit/01ee67d6e0cc5e9d6ae5f07045024a638564fe78. Accessed June 2021

Example of an incorrect parameter value. https://github.com/google/tf-quant-finance/commit/258844720a9bccd326c7b33735f7f81c2d483630. Accessed June 2021

Falleri J-R, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering, pp 313–324

Forward A, Lethbridge TC (2008) A taxonomy of software types to facilitate search and evidence-based software engineering. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, pp 179–191

Github api. https://docs.github.com/en/rest/reference/search. Accessed June 2021

Gulli A, Pal S (2017) Deep learning with Keras. Packt Publishing Ltd

Gu Z, Wu J, Liu J, Zhou M, Gu M (2019) An empirical study on api-misuse bugs in open-source c programs. In: 2019 IEEE 43rd annual computer software and applications conference (COMPSAC), vol 1, pp 11–20. IEEE

Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1110–1121

Institute of Electrical and Electronics Engineers (1987) IEEE Standard Taxonomy for Software Engineering Standards

Islam MdJ (2020) Towards understanding the challenges faced by machine learning software developers and enabling automated solutions

Islam MdJ, Nguyen HA, Pan R, Rajan H (2019) What do developers ask about ml libraries? a large-scale study using stack overflow. arXiv:1906.11940

Islam MdJ, Nguyen G, Pan R, Rajan H (2019) 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, pp 510–520

Islam MdJ, Pan R, Nguyen G, Rajan H (2020) Repairing deep neural networks: fix patterns and challenges. In: 2020 IEEE/ACM 42nd international conference on software engineering (ICSE), pp 1135–1146. IEEE

Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T (2014) Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on multimedia, pp 675–678

Just R, Jalali D, Ernst MD (2014) Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, pp 437–440

Kechagia M, Devroey X, Panichella A, Gousios G, van Deursen A (2019) Effective and efficient api misuse detection via exception propagation and search-based testing. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, pp 192–203

Kechagia M, Mechtaev S, Sarro F, Harman M (2021) Evaluating automatic program repair capabilities to repair api misuses. IEEE Trans Softw Eng
Kuutti S, Bowden R, Jin Y, Barber P, Fallah S (2020) A survey of deep learning applications to autonomous vehicle control. IEEE Trans Intell Trans Syst 22(2):712–733
Kwasnik BH (1999) The role of classification in knowledge representation and discovery
Lamothe M, Guéhéneuc Y-G, Shang W (2021) A systematic review of api evolution literature. ACM Comput Surv (CSUR) 54(8):1–36
Lamothe M, Li H, Shang W (2021) Assisting example-based api misuse detection via complementary artificial examples. IEEE Trans Softw Eng
Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics, pp 159–174
Li X, Jiang J, Benton S, Xiong Y, Zhang L (2021) A large-scale study on api misuses in the wild. In: 2021 14th IEEE conference on software testing, verification and validation (ICST), pp 241–252. IEEE
Liu Y, Liu G, Zhang Q (2019) Deep learning and medical diagnosis. Lancet 394(10210):1709–1710
Liu K, Kim D, Koyuncu A, Li L, Bissyandé TF, Le Traon Y (2018) A closer look at real-world patches. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), p 275–286. IEEE
Mama R (2021) Example of a misused api with incorrect parameter. https://github.com/Rayhane-mamah/Tacotron-2/commit/0ae2901b428afd4127272154b71705e2799a484d. Accessed June 2021
Mamah R (2023) The example of inner api misuse in dl application. https://github.com/Rayhane-mamah/Tacotron-2/commit/fb5564b7584ae0dc62ffecaa89d463ff24a3c251. Accessed Aug 2023
McHugh ML (2012) Interrater reliability: the kappa statistic. Biochem Med 22(3):276–282
Meijer E (2018) Behind every great deep learning framework is an even greater programming languages concept (keynote). In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 1–1
mypy. https://github.com/python/mypy. Accessed June 2021
Nielebock S, Heumüller R, Schott KM, Ortmeier F (2020) Guided pattern mining for api misuse detection by change-based code analysis. arXiv:2008.00277
Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A (2017) Automatic differentiation in pytorch
pylint. https://github.com/PyCQA/pylint. Accessed June 2021
pyre-check. https://github.com/facebook/pyre-check. Accessed June 2021
pyright. https://github.com/microsoft/pyright/. Accessed June 2021
Python standard library. https://docs.python.org/3/library/. Accessed June 2021
Ren X, Ye X, Xing Z, Xia X, Xu X, Zhu L, Sun J (2020) Api-misuse detection driven by fine-grained api-constraint knowledge graph. In: 2020 35th IEEE/ACM international conference on automated software engineering (ASE), pp 461–472. IEEE
Scalabrino S, Bavota G, Linares-Vásquez M, Lanza M, Oliveto R (2019) Data-driven solutions to detect api compatibility issues in android: an empirical study. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp 288–298. IEEE
Shen Q, Ma H, Chen J, Tian Y, Cheung S-C, Chen X (2021) A comprehensive study of deep learning compiler bugs. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 968–980
Šmite D, Wohlin C, Galviņa Z, Prikladnicki R (2014) An empirically based terminology and taxonomy for global software engineering. Empir Softw Eng 19(1):105–153
Svyatkovskiy A, Deng SK, Fu S, Sundaresan N (2020) Intellicode compose: code generation using transformer. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 1433–1443
Tensorflow repositories in githubs. https://github.com/search?q=tensorflow&type=. Accessed June 2021
The manual verification results for api bugs provided by Islam et al. https://zenodo.org/record/8302351. Accessed Aug 2023
Unterkalmsteiner M, Feldt R, Gorschek T (2014) A taxonomy for requirements engineering and software test alignment. ACM Trans Softw Engi Methodol (TOSEM) 23(2):1–38
Usman M, Britto R, Börstler J, Mendes E (2017) Taxonomies in software engineering: a systematic mapping study and a revised taxonomy development method. Inf Softw Technol 85:43–59
Usman M, Gopinath D, Sun Y, Noller Y, Păsăreanu CS (2021) Nn repair: constraint-based repair of neural network classifiers. In: Computer aided verification: 33rd international conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33, pp 3–25. Springer
Vélez TC, Khatchadourian R, Bagherzadeh M, Raja A (2022) Challenges in migrating imperative deep learning programs to graph execution: an empirical study. In: Proceedings of the 19th international conference on mining software repositories, pp 469–481

Wan C, Liu S, Hoffmann H, Maire M, Lu S (2021) Are machine learning cloud apis used correctly? In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE), pp 125–137. IEEE

Wardat M, Cruz BD, Le W, Rajan H (2022) Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In: Proceedings of the 44th international conference on software engineering, pp 561–572

Wardat M, Le W, Rajan H (2021) Deeplocalize: fault localization for deep neural networks. In 2021 IEEE/ACM 43rd international conference on software engineering (ICSE), p 251–262. IEEE

Wen M, Liu Y, Wu R, Xie X, Cheung S-C, Su Z (2019) Exposing library api misuses via mutation analysis. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), pp 866–877. IEEE

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media

Wu D, Shen B, Chen Y (2021) An empirical study on tensor shape faults in deep learning systems. arXiv:2106.02887

Yan M, Chen J, Zhang X, Tan L, Wang G, Wang Z (2021) Exposing numerical bugs in deep learning via gradient back-propagation. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 627–638

Yang Y, Xia X, Lo D, Grundy J (2020) A survey on deep learning for software engineering. arXiv:2011.14597

Yu B, Qi H, Guo Q, Juefei-Xu F, Xie X, Ma L, Zhao J (2021) Deeprepair: style-guided repairing for deep neural networks in the real-world operational environment. IEEE Trans Reliab 71(4):1401–1416

Zar JH (2005) Spearman rank correlation. Encyclopedia of Biostatistics, 7

Zhang Y, Chen Y, Cheung S-C, Xiong Y, Zhang L (2018) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 129–140

Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An empirical study of common challenges in developing deep learning applications. In: 2019 IEEE 30th international symposium on software reliability engineering (ISSRE), pp 104–115. IEEE

Zhang T, Upadhyaya G, Reinhardt A, Rajan H, Kim M (2018) Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow. In: Proceedings of the 40th international conference on software engineering, pp 886–896

Zhang T, Upadhyaya G, Reinhardt A, Rajan H, Kim M (2018) Are online code examples reliable? an empirical study of api misuse on stack overflow. In: International conference on software engineering (ICSE), vol 10

Zhang R, Xiao W, Zhang H, Liu Y, Lin H, Yang M (2020) An empirical study on program failures of deep learning jobs. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1159–1170

Zhong H, Su Z (2015) An empirical study on real bug fixes. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1, pp 913–923. IEEE

Deheng Yang is currently a Ph.D. student at National University of Defense Technology, under the supervision of Dr. Xiaoguang Mao. He received the BA in computer science and technology from the National University of Defense Technology. His research interests include fault localization, automated program repair, etc.

Kui Liu is a research scientist in the software engineering application technology lab at Huawei, China. His research interests focus on Intelligent Software Engineering, including software analysis, software testing, code review, code generation and comprehension.

Yan Lei received the BA, MA and Ph.D. degrees in computer science and technology, all from the National University of Defense Technology, China. He is an associate professor at the School of Big Data & Software Engineering in Chongqing University, China. His research interests include fault localization, program repair, program slicing, etc.

Li Li is a Professor of Software Engineering at Beihang University, China. Before that, he was a Senior Lecturer and ARC DECRA Fellow at Monash University, Australia. He received his PhD degree in computer science from the University of Luxembourg in 2016. Li has awarded the MSR 2023 Ric Holt Early Career Achievement Award and has been ranked as the top-5 most impactful early-stage SE researchers in the world by two continuous bibliometric assessments of SE scholars concerning papers published from 2010 to 2017 and from 2013 to 2020, respectively. He has published over 150 research papers at prestigious conferences such as ICSE, ESEC/FSE, ASE, ISSTA, POPL, PLDI, WWW, and prestigious journals such as ACM TOSEM, IEEE TSE, TIFS, TDSC. His publications have received multiple Best Paper Awards, including an ACM SIGPLAN Distinguished Paper Award in 2021, a Best Student Paper Award at The Web Conference in 2020, an ACM SIGSOFT Distinguished Paper Award in 2018, an MSR FOSS (Free, Open-Source Software) Impact Paper Award in 2018, and a Best Paper Award at SANER-ERA 2016. He is an active member of the software engineering and security community, serving as the Associate Editor for the ACM Computing Survey journal and reviewers for many top-tier conferences and journals such as ASE, ICSME, ISSRE, SANER, MSR, TSE, TOSEM, TIFS, TDSC, TOPS, EMSE, JSS, IST, etc.

Huan Xie is a PhD student advised by Professor Yan Lei at Chongqing University. He received his B.S. degree in software engineering from Chongqing University. His research interests include software testing, fault localization, and deep learning testing.

Chunyan Liu is a master's student under the supervision of Professor Yan Lei of Chongqing University. Her research interests include software testing, fault localization, and deep learning testing.

Zhenyu Wang is a PhD candidate at the College of Computer Science in Chongqing University. He received his bachelor's degree in School of Big Data and Software Engineering from Chongqing University in 2020. His research interests include reinforcement learning, edge intelligence.

Xiaoguang Mao is a professor at College of Computer, National University of Defense Technology, China. His research interests include high confidence software, software development methodology, software assurance, software service engineering, etc.

Tegawendé F. Bissyandé is research scientist with the Interdisciplinary Center for Security, Reliability and Trust (SnT) at the University of Luxembourg. He holds a PhD in computer from the Université de Bordeaux in 2013, and an engineering degree (MSc) from ENSEIRB. His research interests are in debugging, including bug localization and program repair, as well as code search, including code clone detection and code classification.

Authors and Affiliations

Deheng Yang[1] · Kui Liu[2] · Yan Lei[3,4] · Li Li[5] · Huan Xie[3,4] · Chunyan Liu[3,4] · Zhenyu Wang[3] · Xiaoguang Mao[1] · Tegawendé F. Bissyandé[6,7]

Deheng Yang
yangdeheng13@nudt.edu.cn

Kui Liu
brucekuiliu@gmail.com

Li Li
lilicoding@ieee.org

Huan Xie
huanxie@cqu.edu.cn

Chunyan Liu
chunyanliu@cqu.edu.cn

Zhenyu Wang
zhenyuwang@cqu.edu.cn

Xiaoguang Mao
xgmao@nudt.edu.cn

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

1 College of Computer, National University of Defense Technology, Changsha, China

2 Huawei Software Engineering Application Technology Lab, Ningbo, China

3 School of Big Data and Software Engineering, Chongqing University, Chongqing, China

4 Peng Cheng Laboratory, ShenZhen, China

5 School of Big Data and Software Engineering, Chongqing University, Chongqing, China

6 Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg, Luxembourg

7 School of Software, Beihang University, Beijing, China