Understanding the Bug Characteristics and Fix Strategies of Federated Learning Systems

Xiaohu Du*†
Huazhong University of Science and Technology, China
xhdu@hust.edu.cn

Xiao Chen
The Hong Kong University of Science and Technology, China
xchenfu@connect.ust.hk

Jialun Cao‡
The Hong Kong University of Science and Technology, China
jcaoap@cse.ust.hk

Ming Wen*†§
Huazhong University of Science and Technology, China
mwenaa@hust.edu.cn

Shing-Chi Cheung‡
The Hong Kong University of Science and Technology, China
scc@cse.ust.hk

Hai Jin*¶
Huazhong University of Science and Technology, China
hjin@hust.edu.cn

ABSTRACT

Federated learning (FL) is an emerging machine learning paradigm that aims to address the problem of isolated data islands. To preserve privacy, FL allows machine learning models and deep neural networks to be trained from decentralized data kept privately at individual devices. FL has been increasingly adopted in mission-critical fields such as finance and healthcare. However, bugs in FL systems are inevitable and may result in catastrophic consequences such as financial loss, inappropriate medical decision, and violation of data privacy ordinance. While many recent studies were conducted to understand the bugs in machine learning systems, there is no existing study to characterize the bugs arising from the unique nature of FL systems. To fill the gap, we collected 395 real bugs from six popular FL frameworks (Tensorflow Federated, PySyft, FATE, Flower, PaddleFL, and Fedlearner) in GitHub and StackOverflow, and then manually analyzed their symptoms and impacts, prone stages, root causes, and fix strategies. Furthermore, we report a series of findings and actionable implications that can potentially facilitate the detection of FL bugs.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → **Empirical software validation**; • **Computing methodologies** → **Artificial intelligence**.

*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, HUST, Wuhan, 430074, China
†Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
‡Guangzhou HKUST Fok Ying Tung Research Institute
§Corresponding author
¶Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616347

KEYWORDS

Federated Learning, Bug Characteristics, Empirical Study

ACM Reference Format:
Xiaohu Du, Xiao Chen, Jialun Cao, Ming Wen, Shing-Chi Cheung, and Hai Jin. 2023. Understanding the Bug Characteristics and Fix Strategies of Federated Learning Systems. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616347

1 INTRODUCTION

Federated learning (FL) [20] decentralizes the training process of a deep learning model over a network of edge devices and servers without compromising data privacy [14]. Benefited from such advantages, there emerges many popular FL frameworks over the years, including PySyft[1], *Tensorflow Federated* (TFF)[2], FATE[3] and etc. Applications built upon such frameworks have been widely applied to various safety-critical areas, including autonomous driving [16], finance [19], medicine [29], transportation [17, 25], health care [31], robotics [18], and blockchain [15].

As FL systems are increasingly deployed for mission-critical applications, their reliability is becoming a significant concern. Like other software, FL systems also suffer from quality issues. Our collected data from *GitHub* and *StackOverflow* reveal that FL bugs are pervasive, and sometimes may lead to catastrophic consequences. For example, a bug of missing encryption exposed private local data to public, leading to data leakage [24] and attack. Other examples happened in PySyft and FATE (i.e., `#issue1502` and `#issue869` respectively), where the encryption for the gradient and loss were missing. Under such circumstances, attackers could easily decode the confidential data [24], and manipulate them maliciously.

However, the characteristics (e.g., symptoms, root causes, and fixing strategies) of FL bugs are not well understood. While there are earlier researches conducted to characterize the bugs on *machine learning* (ML) and *deep learning* (DL) systems [11, 12, 36, 37], no similar studies have been performed to learn about the characteristics of FL bugs. To bridge the gap, this paper presents the first comprehensive study on characterizing and understanding the bugs in FL systems. Specifically, we investigate the six most widely used FL

[1]https://github.com/OpenMined/PySyft
[2]https://github.com/tensorflow/federated
[3]https://github.com/FederatedAI/FATE

frameworks, which are PySyft, TFF, FATE, Fedlearner[4], PaddleFL[5], and Flower[6]. In particular, our study aims to provide a systematic understanding towards various bugs when constructing FL systems. Note that the root cause of an FL anomaly reported in *GitHub* or discussed at *StackOverflow* may concern program code, data, or execution environments. Our study considers all such anomalies, and refers to them as *bugs*, following the practice as adopted by existing studies [12, 36]. To characterize FL bugs, our study collects 192 issues and 143 pull requests from *GitHub*, together with 60 instances from *StackOverflow*. We analyze these bugs qualitatively and quantitatively from four dimensions, including symptoms and impacts, prone stages, root causes, and fixing strategies.

Our study has led to various interesting findings. For example, we find that one third of the collected bugs' root causes are FL specific (e.g., interaction issues among clients/server and improper security mechanisms), and they are distributed across various implementation stages of an FL system. We summarize the root causes into seven major categories and 28 sub-categories. The largest categories are *Incorrect FL Algorithm Implementation* and *Improper Interaction Initialization/Establish*, which are specific to FL. ML/DL or data related root causes are also common, such as *Incorrect Tensor-related Implementation* and *Incorrect Data Type, Shape & Format* respectively. As for how to fix such bugs, we summarize 19 different strategies categorized into 4 main types. More importantly, we observe new fixing strategies that have not been observed for repairing DL bugs by existing studies [12, 13]. For instance, the strategy of *Modify Configuration* and *Add/Enhance Encryption Steps*, which concern mechanisms unique to FL, are revealed for the first time. Finally, our study also provides actionable advice and implications for practitioners to detect and debug FL bugs. To summarize, this study makes the following major contributions:

- **Originality:** To our best knowledge, we are the first to perform comprehensive studies to characterize FL bugs.
- **Extensive Study:** We collect 335 issues from *GitHub* (i.e., 192 issues and 143 pull requests) and 60 instances from *StackOverflow* covering six popular large-scale FL frameworks, and further perform extensive studies to understand the bug symptoms, root causes, and fix strategies.
- **Empirical Findings:** Our empirical study reveals new findings. Specifically, we find that over half of the bugs are related to FL characteristics. We also observe and summarize common root causes and fix strategies that are specific to the FL mechanisms.
- **Dataset:** We release our collected dataset and the experimental results. The benchmark dataset and results can benefit both practitioners and researchers in conducting future researches. They are available at: **https://github.com/CGCL-codes/FL_Bug_Study**.

2 BACKGROUND

2.1 General Workflow of Federated Learning

FL is an ML paradigm where multiple clients (e.g., mobile devices) collaboratively train a model using their private data under the orchestration of a central server (e.g., service provider). FL can be horizontal (i.e., cross-silo) or vertical (i.e., cross-device) according to the way the data is split [14, 35]. In horizontal FL, a client holds different data instances regarding the same feature set. In vertical FL, a client holds different feature set of the same set of data instances. Despite such differences, the two FL paradigms are subject to the same data processing and exchange protocols. In the following, we illustrate the workflow of horizontal FL. A similar workflow is prescribed by vertical FL.

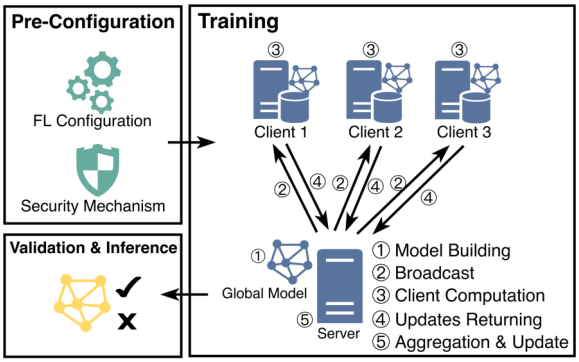


Figure 1: The General Workflow of FL Systems

Figure 1 shows three main stages in the workflow of FL. (1) **Pre-Configuration.** This stage sets up necessary configurations for FL, including the internet IP, port numbers, encryption algorithms [30], and verifiable secret sharing protocols [7]. Such configuration enables secure interactions between clients and servers. (2) **Training.** This stage differs from its centralized ML counterpart. A global model is initialized either from scratch or by loading a pre-trained model (① **Model Building**). The model is propagated to each client (② **Broadcast**). Each client trains the model using its local data (③ **Client Computation**). After the training, it sends the model information (e.g., gradients, thresholds, or other encrypted intermediate parameters according to the protocols) back to the server (④ **Updates Returning**). The server aggregates the received intermediate information and updates the global model following the underlying FL algorithms such as FedAvg [20] and Secure Aggregation [2] accordingly (⑤ **Aggregation and Update**). Steps ② to ⑤ repeat until model converges or reach a termination condition. (3) **Validation and Inference.** After a global ML model has been trained, it can be used for inference after validation.

FL variants are proposed to serve different application scenarios. For example, decentralized FL [26] (where the clients could coordinate themselves without the orchestration of the central server) was proposed to prevent single-point failure. Heterogeneous FL [34] is more suitable to handle heterogeneous clients such as mobile phones and IoT devices. These FL variants were specific to certain application scenarios. In this paper, we present our methodology based on the workflow of general FL rather than a specific variant.

The uniqueness of FL, such as its distributed data, extensive client-server interactions, and strict data privacy protection, presents it with challenges/bugs that traditional ML and DL have not encountered. Such differences could potentially lead to bugs in the FL system that are unprecedented in DL systems. Therefore, addressing these challenges requires an in-depth understanding towards FL's unique mechanism. These bugs and corresponding fix strategies will be discussed in this paper.

[4] https://github.com/bytedance/fedlearner
[5] https://github.com/PaddlePaddle/PaddleFL
[6] https://github.com/adap/flower

2.2 Popular FL Frameworks

Due to their intrinsic complexity, FL systems are mostly built on top of FL frameworks such as PySyft, TFF, and FATE. Besides the libraries supporting the various FL phases, FL frameworks also provide encryption algorithms to assure the security and reliability. For instance, FATE provides security supports such as SecureBoost [6] and Hetero-LR [8], which encrypt the entity-aligned data and gradients in training with the same level of accuracy as the non-privacy-preserving approach. In addition, since many core FL algorithms (e.g., LSTM [22] and BERT [10]) are similar to those in DL, FL frameworks are commonly built on existing DL frameworks. Indeed, various functionalities in TFF and PySyft are implemented based on Tensorflow and PyTorch, respectively. The six popular FL frameworks mentioned above are well-received by FL application developers. For instance, PySyft has received more than 8,000 stars. Meanwhile, bug issues occur frequently as these frameworks evolve frequently to maintain their competitiveness. For instance, there are over 1,278 issues in PySyft, of which around 444 (34.74%) are waiting to be resolved.[7] In view of the increasing prevalence of FL frameworks and the impact of their bug issues, we are motivated to conduct a systematic study of the bug characteristics and their fixes in major FL frameworks.

3 METHODOLOGY

3.1 Data Collection

Since there is no related benchmark dataset available, we construct it by ourselves from two sources: *GitHub* and *StackOverflow*. Over *GitHub*, we collect data from six popular FL frameworks (i.e., according to their *forks* and *stars*), which are PySyft, TFF, Fedlearner, PaddleFL, FATE, and Flower. Over *StackOverflow*, we collect data based on FL-related labels and keywords. The following describes our data collection process.

3.1.1 Data Collection from GitHub. We collect data from the issues and pull requests (PRs) at *GitHub* in three steps.

Step 1: Data collection by label and bug keywords. We define a list named *bug_keywords* and utilize them in the title to query bug-related instances, which contains keywords related to bugs referring to existing studies [11, 37]. Specifically, it contains: *fix, broken, solve, problem, bug, defect, error, incorrect, unsuccessful, wrong, fault, fail, crash, nan, and inf.* We check the labels of all issues and PRs of the six frameworks. First, for the instance whose label is "bug", we directly select it in our dataset. Second, we focus on those instances without labels (i.e., the label is empty or is "cla:yes" or "cla:no" in the TFF framework since they are automatically generated). Such empty labels cannot reflect whether the instance is bug-related or not. Therefore, we further check them against *bug_keywords* to ensure that our bug collection is comprehensive. Noted that the matching is case-insensitive, and partial matching is allowed (e.g., "keyError" matches the keyword "error").

Step 2: Data filtering by irrelevant keywords. To reduce false positives in data collection, we define a list named *irrelevant_keywords* which helps us to exclude those issues or PRs that are not related to bugs. Specifically, it contains *install, build, refactor, rename, typo* since they are common words that are less likely to involve real bugs. Also, we add a condition *merged = "true"* when collecting PRs data to ensure it has been officially recognized.

Step 3: Manual identification. Step 2 excludes a large number of instances that are irrelevant to bugs, while it may still contain false positives that are not related to real bugs. Therefore, a further step of manual validation is necessary. Specifically, we remove the PRs that contain over 10 source files, in which the enclosed content might have merged a large number of issues into a single pull request or involve many extra changes that are irrelevant to bug fixing. We remove those instances since it prevents us from understanding the real root causes. At the same time, we exclude those issues that have not been clearly resolved. For example, the question is unanswered or the questioner does not give feedback on whether the answer is correct. We also employ data deduplication to ensure that instances about the same bug are not counted for multiple times in our study. Specifically, when a pull request is linked to an issue, we only keep the issue because it contains more information to analyze the root causes and fix strategies. After manual validation, we finally obtain 192 issues and 143 PRs from *GitHub*. The collected statistics is shown in Table 1.

Table 1: Statistics of FL Frameworks from GitHub

Framework	Total				Collected data	
	Created	Forks	Stars	LOC	Issues	PRs
Pysyft	2017-07	1,833	8,218	3.8m	80	49
FATE	2019-01	1,310	4,371	497.3k	79	46
TFF	2018-12	473	1,901	206.5k	21	26
Flower	2020-02	293	1,136	49.1k	8	5
Fedlearner	2020-01	166	783	209.8k	2	11
PaddleFL	2019-09	102	393	87.0k	2	6
Total	-	-	-	-	192	143

3.1.2 Data Collection from StackOverflow. *StackOverflow* is a well-known Q&A forum that contains many posts related to real FL bugs, and thus we include it for investigation.

Step 1: Data collection by tags. In the first step, we search for tags related to FL on *StackOverflow*, and identify three tags: *tensorflow-federated, federated-learning*, and *pysyft*. In addition to these tags, we require the number of answers to be greater than one since it is difficult to understand questions without answers. The reason why we choose questions with "answers" rather than "accepted answers" is that we observe there are a handful of unaccepted answers that can correctly solve the questions.

Step 2: Data collection by keywords. In this step, we use keywords to search for posts related to FL regardless of the tags of the posts. The relevant keywords are FL-related, which include *federated learning, tensorflow_federated, syft, flwr, paddle_fl, fedlearner, PySyft*, and *PaddleFL*. Be noted that we no longer use FATE, Flower, federated, etc as keywords. It is because that when we search for them individually in *StackOverflow*, the results exceed 500, and almost all of them are false positives since they are both common words. For further verification, we combine them with keywords "*federated learning*" or "*fl*" to search, and the searched real FL bugs are limited. Similar to the first step, we also require the number of answers to be greater than one, and remove those duplicate posts obtained by searching with keywords or searching with tags.

Step 3: Manual identification. Based on the above steps, we collect a total of 247 posts. However, there are still many instances unrelated to FL. Therefore, we further perform manual verification

[7] Issues and PRs that were posted before July 20, 2022, are collected.

to determine if it is an FL bug by checking for the presence of FL-related libraries, keywords, and code with FL characteristics, such as client/server creation, gradient aggregation, etc. In addition, we exclude irrelevant questions including installation and other bug-irrelevant questions. Finally, we obtain 60 posts.

The final dataset comprises 395 instances, which are analyzed in terms of symptoms, bug stages, and root causes. The scale of this dataset is comparable to that of existing works [11, 12], covering six of the most popular FL frameworks, including not only the inherent errors of the FL frameworks themselves but also errors users encounter with these frameworks, thus providing us with a comprehensive viewpoint. We identify the scope of the instances by manually identifying and analyzing the root causes of bugs, and whether the implemented fix strategies are specific to FL frameworks. One important criterion is whether the fix is merged into FL frameworks by the owners in the form of a PR. This analysis helps us determine whether these bugs originate from FL frameworks or from third-party applications that utilize such frameworks. Among all bugs analyzed, we find that 249 instances (accounting for 63.04%) are directly related to FL frameworks. These bugs arise from the inherent design and implementation errors within FL frameworks. Among these, the framework owners have merged 161 of the fix strategies, which then serve as our primary data source for the analysis of bug-fix strategies presented in Section 7. However, there are 88 bugs for which a clear fix strategy in the source code has not been provided, a pull request link is missing, or the fix does not involve a code change. Additionally, 146 instances (accounting for 36.96%) are related to third-party applications using FL frameworks. These bugs are caused by users improperly using or misunderstanding the functionality of FL frameworks while building third-party applications. In conclusion, the comprehensive and extensive data collection serves as a reliable resource for this study.

3.2 Research Questions

This study aims to answer the following research questions.

RQ1: Symptom and Impact. What are the symptoms of FL bugs, and what are their impacts?

RQ2: Bug Stage. Which stages in FL pipeline are more vulnerable to various types of bugs?

RQ3: Root Cause. What are the root causes of FL bugs and which types are more common?

RQ4: Fix Strategy. How long does it take to fix FL bugs? What is the size of the patches to fix FL bugs? Are there any common patterns to fix these bugs?

3.3 Bugs Classification Method

Overall, in order to ensure the rationality and completeness of our taxonomy, our classification method follows prior extensive empirical studies on DL bugs [11, 36]. In particular, we adopt the open coding paradigm [28] of software engineering, using a bottom-up approach to establish top-level categories, followed by a top-down approach for creating subcategories, resulting in a multi-level taxonomy. The three criteria in our study (i.e., *symptoms and impacts*, *bug stages*, and *root causes*) all follow this classification process, among which the *symptoms and impacts* do not further subdivide into subcategories. Additionally, the categories of *fix strategies* are constructed based on their *root causes*. A detailed description of how to perform open coding to complete the taxonomy is as follows.

First, we conduct a comprehensive analysis for each instance. Specifically, issues on *GitHub* and posts on *StackOverflow* are reviewed for full discussions to avoid being misled by partial comments. For pull requests on GitHub, we check the commit notes and the code diffs. If the code diff does not clearly indicate the type of the bug, we further analyze the contextual code or the entire file, till we understand the root cause of the bug.

Next, two authors with expertise in FL independently study each instance, using descriptive texts to categorize them. After recording all the data, we analyze the connections among all the descriptive texts, grouping related or similar bugs together to form the top-level categories. This task is challenging as we have to summarize 395 instances given a few brief titles.

We then check each instance to see if it could be placed under a subcategory of the top-level categories based on its descriptive text. If no existing subcategory is appropriate, we combine the description text summary with its keywords to create a new one. If there is a fitting subcategory, we directly mark the corresponding instance as its subcategory. We dynamically adjust the parent category and subcategory until the final consistent taxonomy is generated. The identification process of top-level categories and subcategories is iterated four times. After each round, we hold meetings to discuss the rationality and completeness of the taxonomy, and further make adjustments accordingly.

Finally, we emphasize our efforts made to avoid subjectivity during the labeling process. The classification of all bugs is first labeled independently by two authors in this study, and then all the classification results are discussed by them together. For instances with inconsistent labels, if they reach a consensus after discussions, the different categories will be adjusted to one category. If there is no consensus, further discussions will be performed involving two additional authors. If there is eventually no consensus, the instance will be classified as *"others"*. The above adopted strategies also follow existing empirical studies on DL bugs [11, 36].

4 SYMPTOMS AND IMPACTS

In this section, we answer **RQ1** via analyzing the symptoms and impacts of FL bugs. The distribution is visualized in Figure 2.

Crash (231/395). It is the most frequent symptom of FL bugs, which manifest as program execution failures, and throwing exceptions or error messages (e.g., `AttributeError`, `ValueError`).

Incorrect Functionality (78/395). The program executes without throwing any exceptions or errors. However, the results are incorrect or presented in an unexpected way.

Deployment Failure (17/395). This symptom indicates that FL framework deployment fails, and unlike Crash, this type of bug occurs before the execution of FL programs. Such issues are mainly caused by the fact that users use the wrong deployment commands, set the wrong environment variables, etc.

Security Risk (9/395). This impact happens when there exist vulnerabilities in the security mechanism, such as MPC-based secure aggregation for FL (e.g., `#issue1502`), which often leads to data leakage. In addition, the symptom of such issues is hard to be detected while the consequence could be serious [24]. For example, in `#issue869`, the server sends its parameter '`self_wx_square`'

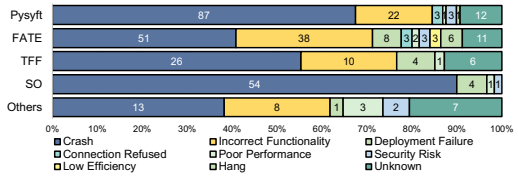


Figure 2: Bug Distribution by Symptoms and Impact

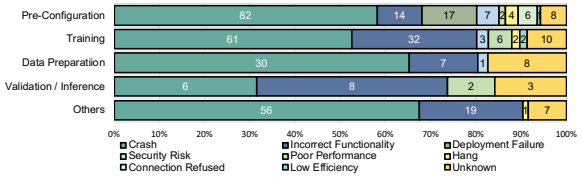


Figure 3: Symptom Distribution by Bug Stages

and '`loss_regular`', to clients without encryption. As a result, clients will know the intermediate loss of the host. This bug can incur information leakage, but it is not fixed until 59 days later.

Poor Performance (8/395). The program executes smoothly and outputs the correct results, but exhibits extremely low accuracy or high loss during model evaluation.

Hang (7/395). The software runs for a long period of time without generating any output.

Connection Refused (6/395). This symptom is that the connection between clients and server fails. For example, before aggregation, the clients need to establish a connection with the server and then transfer information. However, bugs with this type of symptom will interrupt the FL process.

Low Efficiency (3/395). The program execution takes too long. For instance, a slow convergence during the training stage can be caused by either logic errors or incorrect inputs. It is worth noting that, unlike poor performance, low efficiency only consumes time while not compromising the model's performance.

Unknown (36/395). The symptom of some instances is unknown because the information in the discussion is very limited, and most of them are collected from PRs.

Our analysis indicates that *Crash*, *Incorrect Functionality*, and *Deployment Failure* are the top three symptoms that occur most frequently, accounting for 58.48%, 19.75%, and 4.30% respectively. For DL bugs [37], *Error* is the most common symptom, accounting for 69.56%, which is similar to *Crash* in our work. Furthermore, *Low Effectiveness* (i.e., the program exhibits extraordinarily poor accuracy, loss, or other unexpected outputs during the execution phase) in DL bugs accounts for the second largest proportion, which is a combination of our *Incorrect Functionality* and *Poor Performance*. Such results also show a resemblance to our study. Lastly, bugs with symptom *Low Efficiency* account for the smallest proportion of the total number both in our study and the previous studies concerning DL bugs. The above analysis shows that FL and DL bugs share high similarities with respect to symptoms and impacts. However, FL bugs also exhibit symptoms or impacts that DL systems do not have, such as *Connection Refused*, and *Security Risk*.

We further analyze the distribution of these symptoms across different frameworks and *StackOverflow*, as shown in Figure 2. From the figure, it is clear that *Crash* is the most frequently-reported bug symptom. FATE framework contains all eight different types of symptoms, which indicates that FATE bugs are more diverse compared with the other frameworks. The bugs with the fewest symptoms are from *StackOverflow*, which indicates that on developer forums, they prefer to discuss bugs that exhibit clear and observable symptoms (i.e., *Crash* and *Deployment Failure*).

> **Finding 1:** FL bugs share similarity and uniqueness as well with the DL bugs in terms of symptoms.
> **Implication:** A promising way to detect FL bugs with similar symptoms as DL bugs is extending the existing DL bug detection tools. Meanwhile, the unique symptoms of FL bugs further require new detection methods and testing oracle.

5 BUG STAGES

To answer **RQ2**, we classify FL bugs into 5 main types based on the stages following existing studies [4, 5, 12]. And the five stages are further categorized into sub-types based on the FL's certain various stages as shown in Figure 1. Note that the total number of the classified instances in Sections 5 and 6 could go beyond **over 395** , because an instance could belong to more than one category.

Pre-Configuration (141/395). This category of bugs takes the largest proportion among all different categories, which can be further classified into multiple sub-categories. In particular, the bugs in *Security Mechanism* (18/141) are mainly caused by the errors in the FL security protocol that affect the privacy protection mechanism, or errors in the encryption algorithm that lead to data leakage. Such mechanisms are often set in the pre-configuration stage. *FL Configuration* (46/141) bugs are mainly caused by the errors among communication parameters between clients and server. *Dependency* (45/141) bugs occur when there are missing dependencies or incompatible versions between different dependencies. Meanwhile, the *Development* (19/141) of single-machine simulation and multi-machine deployment may encounter certain bugs. In addition, there are a small number (13/141) of environmental-related bugs with limited information but are related to the environment.

Data Preparation (46/395). Bugs occur in the data preparation process can be mainly divided into two categories. The first one is the input data is wrong, including incorrect formats, types, or shapes of original data provided by users. The second one often occurs when the program performs incorrect data loading or preprocessing after taking the correct input data.

Training (119/395). Since the core of FL is learning an optimal ML/DL model, there are considerable bugs that occur when constructing neural networks, decision trees, and other ML algorithms in the stage of *Model Building* (47/119). The bugs in *Broadcast* (5/119) are caused by the wrong implementations when the server sends back model updates to clients, including parameters and weights. Computational errors in the training process of the client side will cause bugs in *Client Computation* (26/119). The wrong aggregation algorithm can lead to bugs in the stage of *Aggregation & Update* (14/119). In addition to the stages as shown in Section 2.1, we also find that a large number of FL bugs are caused by *Transformation & Transfer* (27/119). Such bugs can appear in vertical FL when encrypting and aligning entities between clients, and they can also occur when exchanging encrypted intermediate results between models during model training.

Validation & Inference (19/395). Bugs in this stage are mainly caused by the wrong implementation of evaluation metrics. For

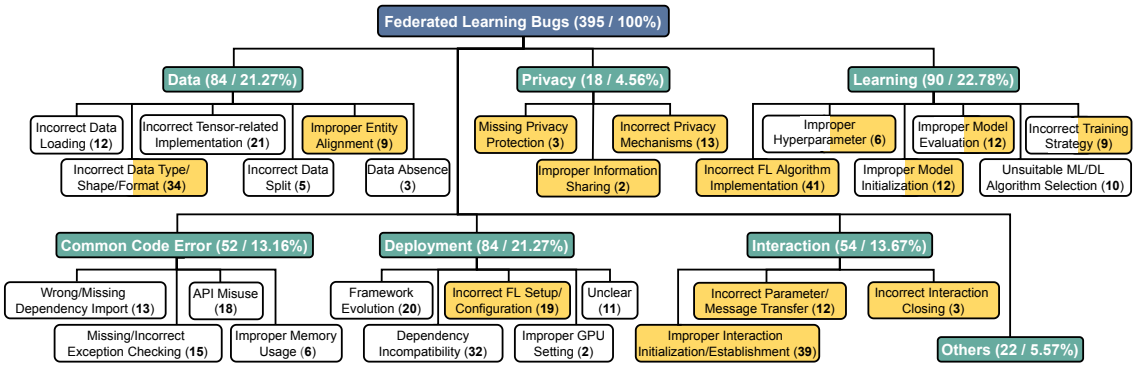


Figure 4: Taxonomy for Root Causes of FL Bugs. We highlight the root causes that are specific to FL systems with , which have not been reported by existing other studies. Boxes of categories that observe FL-specific instances but also contain other cases that are not specific to FL are partially filled. Please see the explanation of each category for details.

example, developers use the ROC curve as an evaluation metric while forgetting to filtering duplicate the data (e.g., #pr1424).

Others (83/395). In addition to the above bugs that often occur in specific stages, some bugs with similar characteristics will appear in multiple stages. *Framework Evolution* (26/83) bugs often occur when the original functions of the FL framework cannot be used after the version is updated. These bugs are more common in PySyft from 2.x to 3.0 and above. *User Interface* (4/83) sometimes encounters bugs when displaying data, but the backend can be run correctly. Also, the fraction of bugs made by novices (*Misuse*) takes up 32% (27/83), and the bugs that lack the necessary information to identify the bug stages are categorized as *Unclear (26/83)*.

We further analyze the distribution of bug symptoms with respect to different bug stages as summarized above, and Figure 3 shows the results. It shows that each category of bugs often results in multiple symptoms. Specifically, we observe all types of symptoms as we summarized for bugs occurring at the stage of *Pre-Configuration*. However, the proportion of each symptom varies across bug stages. For instance, the most common symptom during the *Pre-Configuration* stage is *Crashes*, accounting for 34.9%. The most common symptom during the *Training* stage is *Incorrect Functionality*, accounting for 40.0%. *Deployment Failure* only occurs during *Pre-Configuration*, and *Security Risks* can also occur during the *Data Preparation* stage. Substantial bugs can only be observed after training while the training of FL models is often time-consuming. Thus, detecting and debugging such bugs is crucial.

Finding 2: A large portion of the bugs occur in the stages of *Pre-Configuration* (35.70%) and *Training* (30.13%). In terms of the number of different bug types, the above stages also account for the largest proportion and contain five different types each. Among all sub-categories, bugs in the process of *Model Building* within the *Training* stage take the largest proportion (11.90%).
Implication: Static analysis tools may be useful to detect general configuration bugs [9, 32] in FL systems considering they are effective in detecting DL configuration bugs [21].

6 ROOT CAUSES

In view of the above five bug stages, we further explore the root causes to answer **RQ3**. The taxonomy is organized into seven top categories, and six of which are further divided into subcategories based on Section 3.3. Figure 4 shows the final taxonomy. In particular, we highlight those categories that are **specific** to FL systems, which have not been observed by existing studies related to DL/ML. Be noted that one bug can have multiple root causes, so that the sum of the total instances under each root cause category is greater than the total number of collected bugs.

6.1 Learning (90/395)

This category explores the critical challenges that are encountered during the learning process when constructing FL systems.

Incorrect FL Algorithm Implementation (41/90). FL algorithms are fundamentally distinguished for addressing data privacy (e.g., the *multi-party computation* (MPC) algorithm [3]) and model aggregation (e.g., the Fedavg algorithm [20]). The incorrect implementation of such FL algorithms such as incorrect model parameter updates, and insecure encryption algorithms will lead to a crash (e.g., #so69619028) and unsatisfactory model performance (e.g., #issue93). Besides, FATE supports several advanced FL paradigms such as SecureBoost [6] and Hetero-LR [8]. Yet, its implementation is error-prone. For example, bug issues (e.g., #issue101 and #pr2578) reported the implementation errors in these advanced FL algorithms. Also, there are algorithms used for synchronizing the training steps. The bugs (e.g., #issue540 and #issue27) residing in these algorithms also fall into this category.

Unsuitable ML/DL Algorithm Selection (10/90). This category of bugs is caused by the selection of unsuitable ML/DL algorithms in the learning task. Consequently, the output is unlikely to be satisfactory. For example, the bugs caused by improper DL model architecture such as layer incompatible (e.g., #issue2132), wrong layer type (e.g., #issue3369), incorrect output layer dimension (e.g., #pr1411) also fall into this category.

Incorrect Training Strategy (9/90). This category of bugs is caused by the incorrect setting of the training strategies, such as the *optimizer*. For example, in PySyft, the use of SGD is more recommended than Adam since the framework supports SGD better accordingly to their implementation (e.g., #issue2011).

Improper Model Evaluation (12/90). To evaluate the performance of ML/DL models, metrics such as accuracy and the ROC curve are often used. Bugs in this category are caused by incorrect calculation of evaluation metrics. For example, when developers implement the calculation of the ROC curve metric, the duplicated data are

kept mistakenly. FL frameworks (such as TFF) also provide callable methods to evaluate the client's weight changes on the local dataset and return the results to the server (e.g., #pr515).

Improper Model Initialization (12/90). The parameters in a ML/DL model are randomly or deliberately initialized to obtain a better start of the learning. However, improper initialization may hamper or even prevent models from effective learning. For example, one user mistakenly initializes all the parameters in the deep neural network to be 0, making the model incapable of learning since the output will always be 0 after matrix multiplication (e.g., #so57581345). In addition, the FL frameworks provide a number of FL-specific networks. For example, FATE provides two kinds of Neural Networks for FL: Heterogeneous Neural Networks (for vertical FL) and Homogeneous Neural Networks (for horizontal FL). TFF also provides FL models that can be called directly, and such bugs tend to occur when weights are assigned (e.g., #so72022654).

Improper Hyperparameter (6/90). Hyperparameters such as the learning rate, number of epochs play the same role in FL systems as they do in DL systems, affecting the training effectiveness and efficiency. The settings of such hyperparameters in FL systems are expected to be in line with that for DL systems. Any improper settings on hyperparameters may lead to poor performance. In addition to DL-related hyperparameters, some hyperparameters are specific to FL, such as the number of clients selected per round and the number of local steps per round. Some of the parameters that only appear in the loaded model are FL-specific, such as Homogeneous Neural Networks (e.g., #issue1408).

Finding 3: Around 22.78% of FL bugs are caused by the incorrect learning process. The improper setting of FL algorithms and hyperparameters will cause unsatisfactory performance. Besides, the training strategy selection should also consider the support from the frameworks that are specific to FL mechanisms.
Implication: Since not all FL frameworks are compatible with DL training strategies, the insufficient support for optimizers in FL systems is a limitation that needs to be carefully handled.

6.2 Deployment (84/395)

This category explores the bugs related to the deployment of FL systems, and they are not related to the execution of FL programs.

Dependency Incompatibility (32/84). FL frameworks rely heavily on third-party libraries (e.g., PyTorch and TensorFlow) as dependencies. Incompatibility issues may happen occasionally as dependencies evolve, leading to incorrect functionality or crashes.

Framework Evolution (20/84). Bugs that are related to the evolution of FL frameworks fall into this category. Any incomprehensive consideration about the compatibility between two versions may introduce evolution issues. For example, when the framework evolves with extensive code reorganization, an incompatible interface could bring inconvenience for the users.

Incorrect FL Setup/Configuration (19/84). FL frameworks provide a variety of setup methods, which greatly facilitates the development of FL applications. Such setup methods have more configurations than conventional DL systems, such as multi-client configuration and connectivity. These bugs usually occur before the executions of FL programs, during the setup phase, and involve the establishment of the FL system, the configuration of environmental variables, etc. Such bugs are typically found in configuration files such as JSON, or at the command lines.

Improper GPU Setting (2/84). FL system enables the users to train with GPUs. Yet, configuring the GPU usage improperly will raise errors during execution. For example, the default configuration does not use GPU for training, after modifying certain arguments, the GPUs could work as expected (e.g., #so65434193).

Unclear (11/84). This category covers the bugs that are caused by unclear environmental issues. For example, there is a case that the program resumes normal operation when the developer restarts the server (e.g., #issue632). Yet, from the bug description, the execution environment and the configuration of the operating system cannot be inferred, so no further conclusion could be implied.

Finding 4: A majority of the deployment-related bugs are caused by incompatibility issues (61.90%), occurring across different versions of frameworks and third-party libraries.
Implication: The compatibility of FL systems, especially the compatibility across diverse devices (e.g., mobile phones and IoT devices) needs more carefully check.

6.3 Privacy (18/395)

This category discusses the importance of privacy in FL systems and highlights the critical importance of ensuring robust privacy mechanisms during FL frameworks.

Incorrect Privacy Mechanisms (13/18). Privacy is always the first-order concern in FL [14]. Therefore, FL framework developers provide a variety of mechanisms to protect user privacy, such as SMC, Differential Privacy, and Homomorphic Encryption. Such FL bugs are caused by the flaws in such mechanisms. For instance, an overflow error occurs when the two numbers multiplied by the Paillier algorithm contain fractions when decrypting (e.g., #issue393). In addition, there are some encoder/decoder issues in SPDZ (e.g., #pr2309 and #pr2543), which is an open source library for secure multi-party computation.

Improper Information Sharing (2/18). An important feature of FL is the nature of data independence of each client and server, prohibiting direct data sharing between them. FL trains the model by passing model parameters, utilizing data from all parties while maintaining strict data sharing rules. For instance, in #issue2365, `model.share()` is correct because it only shares model parameters, not the data itself. In contrast, `data.share()` can lead to various issues as it violates the principle of data independence in FL systems. Note that data sharing here does not involve specific privacy protection mechanisms, such as MPC, making it distinct from the category of *Incorrect Privacy Mechanisms*.

Missing Privacy Protection (3/18). Such bugs are more severe than the others since they may directly lead to serious consequences such as data leakage [24]. For instance, in #issue1502, gradients should not be sent directly to the server, and MPC should be employed to safely aggregate gradients. It differs from *Incorrect Privacy Mechanisms* as it represents a complete missing of privacy protection, rather than an incorrect implementation of privacy protection. While *Incorrect Privacy Mechanisms* may not necessarily lead to privacy leaks, they could result in incorrect outcomes.

Finding 5: The encryption mechanism is a unique characteristic of FL systems, while the bugs of missing or incorrect implementation of the encryption algorithm are pervasive in FL systems.
Implication: The implicit symptom protects the encryption-related bugs from being easily detected. While considering the sever security issue it brings about (i.e., data leakage), detection approaches are in need.

6.4 Interaction (54/395)

The category discusses the challenges in the interaction process of FL systems. Ensuring proper interaction procedures, from establishment to closure, is vital for successful constructing FL systems.
Improper Interaction Initialization/Establishment (39/54). These bugs arise during the execution of FL programs, specifically with the correct initialization and establishment of interactions between clients and the server within an established FL environment. This phase includes the assignment of client and server roles and their parameters, as well as the initialization of interaction mechanisms. Bugs of this category are usually found at the code level. Elements such as the setup of IP addresses and host names are crucial during this phase. If the interaction is not properly initialized/established due to incorrect parameters like IP or host name, the interaction will fail. For instance, an incorrect host name setup will lead to an establish failure (e.g., `#issue1535`).
Incorrect Parameter/Message Transfer (12/54). After the interaction is established, the parameters and messages could be transferred. However, incorrect information transfer can also cause various issues such as wrong parameter updates, or the update failure (e.g., `#pr4640` and `#issue2447`).
Incorrect Interaction Closing (3/54). After the transfer finishes, the interaction should be closed properly. Otherwise, it may result in failures of model storage and evaluation (e.g., `#issue540`).

Finding 6: Most (39/54, 72.22%) of the malfunctions between clients and the central server could be attributed to incorrect configurations concerning IPs, the ports, the host name and etc.

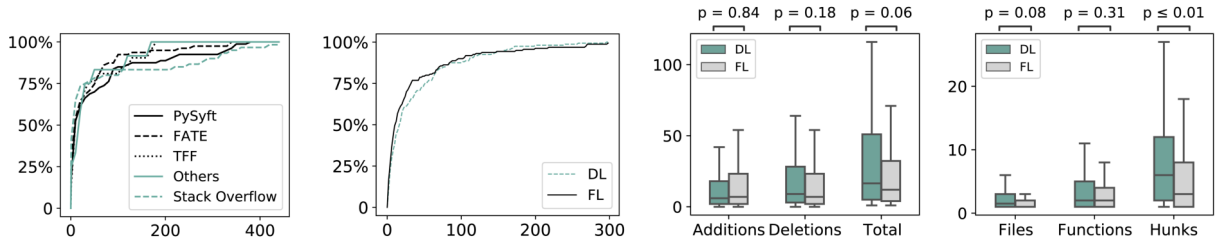
6.5 Data (84/395)

This category includes various bugs related to data in FL systems. It is crucial for FL developers to ensure proper handling of tensors, data types, alignment, loading, splitting, and avoid missing data to ensure the reliability and accuracy of FL models.
Incorrect Tensor-related Implementation (21/84). A tensor in ML/DL/FL represents a multidimensional array. Tensor computations are pervasive and significant in DL models. Thus, the incorrect implementation over tensors (including tensor shape mismatch, tensor unalignment, wrong computation over tensors) is a major root cause, leading to incorrect training results or even crash.
Incorrect Data Type, Shape & Format (34/84). Incorrect type, shape, or format of data is the most major root cause in this category, accounting for over half of the cases. These bugs often explicitly raise exceptions such as "TypeError", "IndexError", and "ValueError". For example, an error will occur if the data type `list` is expected but a `dict` is received (e.g., `#so62332459`). Besides, each client of the FL system needs to share the same label. In a multi-classification task, if the data is uploaded to different clients according to the label category, it will fail when the data labels owned by different roles are different. Also, it fails when the predicted label and the label of the training data are inconsistent. Such inconsistencies include the label name (e.g., `#issue1376`) and label type (e.g., `#issue170` and `#issue1089`).
Improper Entity Alignment (9/84). Entity alignment is utilized to match the identical data instances or features across the datasets stored in clients without exposing the data. For example, an issue regarding the poor model performance turns out to be attributed to the unaligned order of parameters returning back from clients (e.g., `#issue2758`). It is caused by incorrect entity alignment and is a specific bug in FL systems.
Incorrect Data Loading (12/84). This category describes the root causes of wrong implementations in data loading. For example, incorrect file path, duplicate data loading, or data loading function is incorrectly implemented.
Incorrect Data Split (5/84). Before training, the dataset will be split into the training set, validation set, and testing set. Wrong implementations of data split can incur data error. For instance, incorrect data split leads to inconsistent headers (e.g., `#issue887`) in the sub-datasets or empty dataset after splitting (e.g., `#pr1749`).
Data Absence (3/84). A small proportion of bugs are caused by missing data, which are often blamed to novice or careless FL developers who forget to provide the input data.

Finding 7: The unmatched data type, shape, and format accounts for 40.48% (34/84) of the issues in this category, which takes up the largest proportion in this category.
Implication. The correctness and robustness of data processing in FL systems are in desperate need. Also, aligning and processing data while protecting data privacy also poses a challenge to FL developers.

6.6 Common Code Error (52/395)

This category is often observed in traditional software development, and FL systems have not been able to avoid such bugs neither. Addressing and resolving these common code errors is essential for ensuring the successful implementation of FL systems.
API Misuse (18/52). API misuse is caused by the incorrect usage of APIs in FL frameworks or other third-party libraries.
Missing/Incorrect Exception Checking (15/52). Similar to conventional software systems, the exception-checking mechanism helps to detect and handle abnormal inputs, parameter settings, and program behaviors to ensure system quality. The lack of complete exception-checking mechanisms weakens the robustness of FL frameworks. For example, the floating-point numbers cannot be used as random seed (e.g., `#issue701`), exception checking for such illegal input is missing in FATE.
Wrong/Missing Dependency Import (13/52). Before implementing the main functionality of the program, developers need to import necessary dependencies. If the import statements of required dependencies are missing or wrong (e.g., incorrect full-qualified names of dependencies), the program can not be executed properly.
Improper Memory Usage (6/52). Bugs that are caused by incorrect memory usage fall into this category. For example, the program



(a) Lifecycle of FL bugs. (b) Loc of FL bugs. (c) Changed lines of FL bugs. (d) Changed files/functions/hunks.

Figure 5: The Quantitative Study on FL Bugs and Comparison with DL Bugs

does not actively release resources in time, resulting in an abnormal termination (e.g., `#issue2082`). These bugs may lead to low efficiency, program hang, and memory explosion (e.g., `#issue1245`).

Finding 8: Common code errors take up 13.16% (52/395) of the bugs in FL systems. Among them, API misuse and missing/incorrect exception checking are the top two root causes.
Implication. Traditional testing and debugging techniques are still useful to resolve common errors in FL systems.

6.7 Others (22/395)

The bugs whose root causes that are unclear due to the limited information of bug description, and bugs that caused by accident mistakes such as typos are categorized into this group since the analysis on them is insignificant to characterize the bug taxonomy.

7 FIX STRATEGIES

The previous sections have delved into the symptoms, stages, and root causes of bugs in the FL workflow. However, a thorough understanding of how bugs are originated only is insufficient to fully counteract them. It is also crucial to understand how to effectively fix these bugs. In this section, we aim to answer RQ4 by focusing on the process of fixing FL bugs. By combining both a quantitative study on the bug lifecycle and patches as well as a qualitative study on types of fix strategies, we comprehensively dissect and collate how FL bugs are fixed. The quantitative study on the lifecycle includes 192 issues from *GitHub* and 60 posts from *StackOverflow*. For patch analysis, we inspect 161 instances as described in Section 3.1. In addition, to understand the difference in patch sizes between FL and DL systems, we also conduct the statistics over DL bug benchmarks [11, 13] under the same settings. After ruling out instances with inaccessible links, there remain 160 instances. Our qualitative study consolidates four main categories of fix strategies, categorized based on the taxonomy of root causes.

7.1 Quantitative Study

Bug Lifecycle. To understand the lifecycle of FL bugs, we first count the time taken to fix a bug in FL systems. Specifically, we measure the time starting from when an issue has been raised to it has been resolved/closed, and Figure 5(a) shows the results. We can see that bugs of four categories (PySyft, FATE, TFF, and Others) from *GitHub* are often fixed in one day for 18.75%, 30.38%, 14.29%, and 25% of the cases respectively, while such a ratio is 41.67% for bugs from *StackOverflow*. It indicates that bugs reported on *StackOverflow* tend to be resolved quickly. After ten days, bugs from PySyft, FATE, TFF, and *StackOverflow* are fixed by more than half (52.5%, 54.43%, 52.38%, and 65%). There are still many bugs that have not been fixed until 50 days, and their proportions range from 16.66% to 30%. Finally, the rest takes developers a longer time to fix, and certain bugs cannot be correctly resolved over a year. The above results reveal that a proportion of FL bugs can be fixed in time while a significant proportion cannot, which reflects the challenges for developers to fix FL bugs.

Patch Size. To further understand the challenges of fixing FL bugs, we first analyze the linear relationship between different *lines of code* (LOC) and the percentage of fixed bugs, as shown in Figure 5(b). We find that 10 lines can fix 39.13% of DL bugs and 49.69% of FL bugs, and most bugs (74.53% for DL and 79.50% for FL) are fixed within 50 lines. However, 12.42% of DL bugs and 9.94% of FL bugs need over 100 lines to fix. Furthermore, in order to understand whether DL bugs and FL bugs are similar in the distribution of patch sizes, we analyze the distribution of their patches with respect to added lines, deleted lines, and total lines, as shown in Figure 5(c). We find that most FL bugs have more LOC than DL on added lines, and most DL bugs have more LOC than FL on deleted lines and total lines. To test whether such a difference is significant, we perform the Mann-Whitney U-Test [27]. As shown by the p-value, we can observe that the difference is insignificant (p-value > 0.05), which indicates that FL and DL bugs share similar distributions with respect to their patch sizes. We further investigate the distributions of FL patches' locations, in particular in terms of the number of hunks (i.e., a group of contiguous lines that are changed along with contextual unchanged lines [1]), functions and source files modified by the patches. As shown in Figure 5(d), overall, FL patches modify fewer files and functions than DL patches, but this difference is not significant. However, FL patches modify significantly fewer hunks than that of DL patches (p-value ≤ 0.01).

Based on the above qualitative analysis, we can derive the following implications. First, given that there is no statistical difference between the sizes of DL and FL patches (e.g., as reflected by the addition, deletion of code lines), and the fact that repairing DL bugs is challenging [12, 13], it can be inferred that repairing FL bugs will also be non-trivial. Second, the code required to be modified to fix FL bugs tends to be less scattered (e.g., as reflected by the significant smaller amount of modified hunks), which can shed lights on the future detection and repairing of FL bugs. For instance, when designing automated repair techniques for FL bugs, the mutation operators can be applied to those closely related statements or hunks without the necessary to search over a larger search space

across different files. Consequently, the repair effectiveness and efficiency can be boosted.

7.2 Qualitative Study

To study how FL bugs are fixed in practice, we summarize four main categories of fix strategies based on the taxonomy of root causes, among which fix strategies for FL specific bugs are focused from a variety of root cause categories. These fix strategies are primarily summarized from merged *Pull Requests*, adopted *Issues* from developer and posts on *StackOverflow*. We believe that due to their extraction from real-world practices and widespread adoption, they possess a certain level of feasibility and effectiveness. Noted that we also summarize certain fix strategies related to DL mechanisms, but they are similar to those as summarized by existing studies [13]. Therefore, we do not list them due to page limit.

7.2.1 Fix Strategies for FL Specific Bugs. We summarize following strategies for bugs specific to FL.

Modify Configuration. Modifying the configuration such as port or host name when the interaction fails to be built. For example, in some instances, the host is changed from `[::]:8080` to `localhost:8080` to ensure the system work normally.

Add Initialization. The lack of initialization could be fixed by adding an initialization step properly. For example, in PySyft, instances of the server and clients should be initialized and then instantiated before the interaction could be built (e.g., `#issue3002`).

Reorder Initialization The initialization should follow certain orders. For example, in PySyft, the server should be initialized before initializing the clients (e.g., `#issue3774`).

Remove Redundant Initialization. Removing the duplicated initialization code snippets, so that the clients or the server would not be accidentally overloaded multiple times.

Modify Parameter. The construction of FL models is often based on the DL framework, but the parameter settings that work in DL may be wrong in FL models. As shown below, TensorFlow Federated cannot understand Keras models compiled with string arguments. TFF requires that compile() calls on the model be given an instance of `tf.keras.losses.Loss` or `tf.keras.metrics.Metric`.

```
def create_compiled_keras_model():
   ...
   opt = Adam(lr=0.0005)
-  def loss_fn(y_true, y_pred):
      return tf.reduce_mean(tf.keras.losses.
        binary_crossentropy(y_pred, y_true))
-  model.compile(optimizer=opt,loss=loss_fn,metrics=['accuracy'])
+  model.compile(optimizer=opt,
         loss=tf.keras.losses.BinaryCrossentropy(),
         metrics=[tf.keras.metrics.Accuracy()])
   return model
```

Downgrade/Upgrade Framework. If the framework involves substantial changes in the implementation, an upgrade or downgrade are common solutions adopted by users, and also suggestions made by developers (e.g., `#pr1424` and `#issue1377`).

Add/Enhance Encryption Steps. If the encryption step is missed or incompletely added, data security is under concern. In this circumstance, adding or enhancing the encryption step could address the data security risk. For example, FATE missed encrypting the model updates sent from clients to the server, exposing the clients' data the risk of data leakage (e.g., `#pr1424` and `#commit890`), as shown below. They fixed this bug in 3 days after it was revealed.

```
-   self.remote_loss_intermediate(self_wx_square)
+   en_wx_square=cipher_operator.encrypt(self_wx_square)
+   self.remote_loss_intermediate(en_wx_square)
```

7.2.2 Fix Strategies for Bugs Related to Deployment. We summarize the following fix strategies for bugs related to deployment.

Reinstall Dependency of Compatible Version. Reinstalling the incompatible dependencies with the proper versions is a straightforward fix. It may be followed by several iterations, because the newly-installed dependencies may have cascading effects on their underlying dependencies, calling for more dependencies update.

Configure the Environment. By authorizing the proper user permission and configuring the necessary environment variables are useful to fix deployment-related bugs.

Configure GPU Usage. To usage of GPU in FL systems, certain configurations need to be set up properly. For example, to use multiple GPUs, the argument `use_experimental_simulation_loop` needs to set as `True` (e.g., `#so65434193`).

Relaunch the System. FL system involves the interactions and system settings, the modification on some of which may take effect after restarting the system. We thus list it as a fix since it is suggested by both framework developers and users.

7.2.3 Fix Strategies for Common Code Errors. The fix strategies for bugs related to common code errors are summarized as follows.

Optimize Memory Usage. Fixes such as releasing memory periodically, removing repeated operations of calculation, or reducing the redundant iterations are strategies to optimize the memory usage.

Adding/Replacing APIs. Adding an useful API (e.g., `#pr1612`), or replacing an improperly-used API with the correct one (e.g., `#pr826`) in aid of the documentation is a direct fix to the API misuse.

Adding/Removing/Replacing the API Parameters. Another kind of API misuse is providing incorrect parameters to the API. In this case, the effective fix is to add, remove, or replace the parameters with the correct types under the instruction of the documentation.

Add/Modify Exception Checking. Adding or modifying the exception checkers to capture and handle the exceptions enhances the program's robustness. For example, FATE strengthened the condition of an exception checker to prevent the random seed to be the type of floating point (e.g., `#commits694`).

Add/Modify Dependency. If the required dependencies are missing, or wrong dependencies are imported, the straightforward solution is to add or replace the correct ones. For example, a user was expected to use `WebsocketServerWorker`, while he/she mistakenly imported it from `syft.workers`. Instead, the correct dependency resides in `syft` (e.g., `#pr2749`).

7.2.4 Fix Strategies for Data Bugs. The following summarizes the fix strategies for bugs related to data.

Remove Redundant Data Loading. Removing the repeated data downloading and loading process helps to avoid useless overhead. This fix saves much time especially when the dataset is huge, downloading and loading which cost considerable time consumption.

Modify the Data Processing. After the dataset has been loaded, applying a proper process to data avoids the inconsistency of datasets across clients, as well as benefiting the training effectiveness. For example, reordering the features following certain orders consistently over clients ensures the correctness of the data split.

Modify Data Type/Shape/Format. Converting the data type, aligning the data shape with the layer dimension, converting the data format to the desired one are fixes to this type of bug.

Finding 9: We summarize 19 different fix strategies grouped into four main categories. Moreover, new fixing strategies (*Add/Enhance Encryption Steps* and *Modify Configuration*) that are specific to FL systems have been identified.
Implication. New mutation operators are required to be devised when designing automated repair techniques to fix FL bugs. For instance, mutators that can synthesize FL bugs induced by inappropriate interaction (e.g., *changing FL configurations*) or encryption (e.g., *adding encryption methods*) are expected.

8 THREATS TO VALIDITY

This study is subject to several threats. First, **Selection of labels and keywords.** We use labels and keywords to match bug instances during data collection, which might contain potential bias. To mitigate this threat, we select the keywords that are widely used in data collection by existing related works [11, 12, 36, 37]. Second, **Selection of data.** The credibility of data collection may lead to external threats. To mitigate this threat, we take the following measures. First, we select two typical bug sources that are widely used in empirical research: *GitHub* and *StackOverflow*. To make our collected bugs highly relevant to FL, our identification is based on relevant frameworks [11, 36, 37]. To mitigate the bias in framework selection, we select six representative frameworks based on Forks and Stars. Meanwhile, these frameworks cover horizontal and vertical FL scenarios, ensuring the extensiveness and comprehensiveness of the data. To mitigate the bias in over-sampling on identical bugs, we reduce duplicate instances via checking the link relationship between instances. To ensure the correctness of our conclusions, we only study fixed bugs, as unfixed or unconfirmed cases may not be real ones. Third, **Subjectivity of manual labeling.** To mitigate the threat from manual labeling, we strictly follow the method in Section 3.3 to avoid subjectivity through multi-person collaborative labeling and additional validation.

9 RELATED WORK

Distributed/IoT System. The distributed system is a set of autonomous components located on different machines [23]. We compare it with the FL system to illustrate the differences. (1) The distributed system supports resource sharing while the FL system insists on independent client data. (2) Different nodes of a distributed system can coexist and work together while all clients of the FL system work independently. (3) Distributed system is fault-tolerant. However, node exceptions in FL systems may compromise the aggregation algorithm. Due to the above differences, the same design that works fine in a distributed system may induce a bug in FL systems. *Internet of Things* (IoT) enables users to interact with them via the internet directly or through programs encapsulating their behavior and goals [33]. IoT system enables communication between devices through cloud servers, differing from FL systems that require precise server/client initialization to prevent bugs. For the interaction, IoT messages are sent to IoT devices through the cloud, and the rate and order error of these messages will lead to bugs, which is also different from that of FL systems.

Empirical Bug Studies. No existing work is found to characterize the bugs in FL systems while most of the related works are empirical studies focusing on DL systems. Zhang *et al.* [37] investigated the symptoms and root causes of 175 bugs in DL applications built on *TensorFlow*. They further summarized challenges and strategies for bug detection and localization. Islam *et al.* [12] investigated the impacts, types, root causes, and bug-prone stages of bugs in DL softwares collected from *Github* and *StackOverflow* across five popular DL frameworks. They further studied the relationship among bugs in different DL frameworks and summarized the changing trend of two types of bugs. Humbatova *et al.* [11] studied three DL frameworks and collected 375 bugs. Different from previous works, they enriched the taxonomy by interviewing 20 researchers and practitioners in structured interviews, which help them find a variety of additional faults that did not emerge from *StackOverflow* and *GitHub*. Compared to these empirical studies, our work complements them by reviewing FL bugs.

The reason our taxonomy differs from DL taxonomies [5, 11] lies in the following folds. First, FL systems introduce unique mechanisms, such as the interactions between devices and the errors that arise from coordinating multiple devices during deployment. Consequently, we separate these aspects into top-level categories to enable more specific studies. Second, DL bugs appear less frequently among the data we collected. We speculate that this might be caused by the extensive and in-depth studies on DL bugs conducted in many previous works. Therefore, developers manage to sidestep the majority of DL bugs when constructing FL systems. Finally, our taxonomy simplifies DL categories, as they contribute less to understanding the characteristics of bugs specific to FL.

10 CONCLUSION

In this paper, we present a comprehensive study on 395 real bugs of FL systems collected from *GitHub* and *StackOverflow*, and analyze their symptoms, stages where they appear, the root causes, and fix strategies. We find that FL bugs share non-negligible resemblance with ML/DL bugs while one-third of the FL bugs are specific to FL systems. Specifically, 65.82% of FL bugs occur in the stage of pre-configuration and training. Also, the main root causes of FL bugs are improper interactions between clients and server and incorrect FL algorithm implementation. We further summarize 19 different strategies categorized into four main types that are commonly used to fix FL bugs. Finally, our study provides implications for FL system developers to facilitate the secure development of FL applications.

11 DATA AVAILABILITY

The data, source code, and the results of this paper is available at: **https://github.com/CGCL-codes/FL_Bug_Study**.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (Grant No. 62002125 and No. 61932021), the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001), the Hong Kong Research Grant Council/General Research Fund (Grant No. 16205722), the Hong Kong Research Grant Council/Research Impact Fund (Grant No. R5034-18), and the Hong Kong Innovation and Technology Fund (Grant No. MHP/055/19).

REFERENCES

[1] Abdulkareem Alali, Huzefa H. Kagdi, and Jonathan I. Maletic. 2008. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, René L. Krikhaar, Ralf Lämmel, and Chris Verhoef (Eds.). IEEE Computer Society, 182–191. https://doi.org/10.1109/ICPC.2008.24

[2] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1175–1191. https://doi.org/10.1145/3133956.3133982

[3] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. 1996. Adaptively Secure Multi-Party Computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 639–648. https://doi.org/10.1145/237814.238015

[4] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, and Xin Peng. 2021. Characterizing Performance Bugs in Deep Learning Systems. *CoRR* abs/2112.01771 (2021). arXiv:2112.01771 https://arxiv.org/abs/2112.01771

[5] Junjie Chen, Yihua Liang, Qingchao Shen, and Jiajun Jiang. 2022. Toward Understanding Deep Learning Framework Bugs. *CoRR* abs/2203.04026 (2022). https://doi.org/10.48550/arXiv.2203.04026 arXiv:2203.04026

[6] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Dimitrios Papadopoulos, and Qiang Yang. 2021. SecureBoost: A Lossless Federated Learning Framework. *IEEE Intell. Syst.* 36, 6 (2021), 87–98. https://doi.org/10.1109/MIS.2021.3082561

[7] Paul Feldman. 1987. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*. IEEE Computer Society, 427–437. https://doi.org/10.1109/SFCS.1987.4

[8] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. 2017. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *CoRR* abs/1711.10677 (2017). arXiv:1711.10677 http://arxiv.org/abs/1711.10677

[9] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. CP-Detector: Using Configuration-related Performance Properties to Expose Performance Bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 623–634. https://doi.org/10.1145/3324884.3416531

[10] Zhiqi Huang, Fenglin Liu, and Yuexian Zou. 2020. Federated Learning for Spoken Language Understanding. In *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, Donia Scott, Núria Bel, and Chengqing Zong (Eds.). International Committee on Computational Linguistics, 3467–3478. https://doi.org/10.18653/v1/2020.coling-main.310

[11] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1110–1121. https://doi.org/10.1145/3377811.3380395

[12] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 510–520. https://doi.org/10.1145/3338906.3338955

[13] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: fix patterns and challenges. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1135–1146. https://doi.org/10.1145/3377811.3380378

[14] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. 2021. Advances and Open Problems in Federated Learning. *Found. Trends Mach. Learn.* 14, 1-2 (2021), 1–210. https://doi.org/10.1561/2200000083

[15] Hyesung Kim, Jihong Park, Mehdi Bennis, and Seong-Lyun Kim. 2020. Blockchained On-Device Federated Learning. *IEEE Commun. Lett.* 24, 6 (2020), 1279–1283. https://doi.org/10.1109/LCOMM.2019.2921755

[16] Xinle Liang, Yang Liu, Tianjian Chen, Ming Liu, and Qiang Yang. 2019. Federated Transfer Reinforcement Learning for Autonomous Driving. *CoRR* abs/1910.06001 (2019). arXiv:1910.06001 http://arxiv.org/abs/1910.06001

[17] Wei Yang Bryan Lim, Jianqiang Huang, Zehui Xiong, Jiawen Kang, Dusit Niyato, Xian-Sheng Hua, Cyril Leung, and Chunyan Miao. 2021. Towards Federated Learning in UAV-Enabled Internet of Vehicles: A Multi-Dimensional Contract-Matching Approach. *IEEE Trans. Intell. Transp. Syst.* 22, 8 (2021), 5140–5154. https://doi.org/10.1109/TITS.2021.3056341

[18] Boyi Liu, Lujia Wang, and Ming Liu. 2019. Lifelong Federated Reinforcement Learning: A Learning Architecture for Navigation in Cloud Robotic Systems. *IEEE Robotics Autom. Lett.* 4, 4 (2019), 4555–4562. https://doi.org/10.1109/LRA.2019.2931179

[19] Yuan Liu, Zhengpeng Ai, Shuai Sun, Shuangfeng Zhang, Zelei Liu, and Han Yu. 2020. FedCoin: A Peer-to-Peer Payment System for Federated Learning. In *Federated Learning - Privacy and Incentive*, Qiang Yang, Lixin Fan, and Han Yu (Eds.). Lecture Notes in Computer Science, Vol. 12500. Springer, 125–138. https://doi.org/10.1007/978-3-030-63076-8_9

[20] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.). PMLR, 1273–1282. http://proceedings.mlr.press/v54/mcmahan17a.html

[21] Mohammad Mehdi Morovati, Amin Nikanjam, Foutse Khomh, and Zhen Ming (Jack) Jiang. 2023. Bugs in machine learning-based systems: a faultload benchmark. *Empir. Softw. Eng.* 28, 3 (2023), 62. https://doi.org/10.1007/s10664-023-10291-1

[22] Viraaji Mothukuri, Prachi Khare, Reza M. Parizi, Seyedamin Pouriyeh, Ali Dehghantanha, and Gautam Srivastava. 2022. Federated-Learning-Based Anomaly Detection for IoT Security Attacks. *IEEE Internet Things J.* 9, 4 (2022), 2545–2554. https://doi.org/10.1109/JIOT.2021.3077803

[23] Nitin Naik. 2021. Demystifying Properties of Distributed Systems. In *Proceedings of the IEEE International Symposium on Systems Engineering, ISSE 2021, Vienna, Austria, September 13 - October 13, 2021*. IEEE, 1–8. https://doi.org/10.1109/ISSE51541.2021.9582515

[24] Milad Nasr, Reza Shokri, and Amir Houmansadr. 2019. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 739–753. https://doi.org/10.1109/SP.2019.00065

[25] Jason Posner, Lewis Tseng, Moayad Aloqaily, and Yaser Jararweh. 2021. Federated Learning in Vehicular Networks: Opportunities and Solutions. *IEEE Netw.* 35, 2 (2021), 152–159. https://doi.org/10.1109/MNET.011.2000430

[26] Abhijit Guha Roy, Shayan Siddiqui, Sebastian Pölsterl, Nassir Navab, and Christian Wachinger. 2019. BrainTorrent: A Peer-to-Peer Environment for Decentralized Federated Learning. *CoRR* abs/1905.06731 (2019). arXiv:1905.06731 http://arxiv.org/abs/1905.06731

[27] Graeme D. Ruxton. 2006. The unequal variance t-test is an underused alternative to Student's t-test and the Mann–Whitney U test. *Behavioral Ecology* 17, 4 (05 2006), 688–690. https://doi.org/10.1093/beheco/ark016

[28] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[29] Micah J. Sheller, G. Anthony Reina, Brandon Edwards, Jason Martin, and Spyridon Bakas. 2018. Multi-institutional Deep Learning Modeling Without Sharing Patient Data: A Feasibility Study on Brain Tumor Segmentation. In *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries - 4th International Workshop, BrainLes 2018, Held in Conjunction with MICCAI 2018, Granada, Spain, September 16, 2018, Revised Selected Papers, Part I (Lecture Notes in Computer Science, Vol. 11383)*, Alessandro Crimi, Spyridon Bakas, Hugo J. Kuijf, Farahani Keyvan, Mauricio Reyes, and Theo van Walsum (Eds.). Springer, 92–104. https://doi.org/10.1007/978-3-030-11723-8_9

[30] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 4 (2018), 769–790. https://doi.org/10.1109/TPAMI.2017.2699960

[31] Jie Xu, Benjamin S. Glicksberg, Chang Su, Peter B. Walker, Jiang Bian, and Fei Wang. 2021. Federated Learning for Healthcare Informatics. *J. Heal. Informatics Res.* 5, 1 (2021), 1–19. https://doi.org/10.1007/s41666-020-00082-4

[32] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4,*

2016, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 619–634. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu

[33] Teng Xu, James B. Wendt, and Miodrag Potkonjak. 2014. Security of IoT systems: design challenges and opportunities. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, Yao-Wen Chang (Ed.). IEEE, 417–423. https://doi.org/10.1109/ICCAD.2014.7001385

[34] Zirui Xu, Fuxun Yu, Jinjun Xiong, and Xiang Chen. 2021. Helios: Heterogeneity-Aware Federated Learning with Dynamically Balanced Collaboration. In *Proceedings of the 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 997–1002. https://doi.org/10.1109/DAC18074.2021.9586241

[35] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *ACM Trans. Intell. Syst. Technol.* 10, 2 (2019), 12:1–12:19. https://doi.org/10.1145/3298981

[36] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1159–1170. https://doi.org/10.1145/3377811.3380362

[37] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. https://doi.org/10.1145/3213846.3213866

Received 2023-03-02; accepted 2023-07-27