2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)

FedDebug: Systematic Debugging for Federated Learning Applications

Waris Gill
Computer Science Department
Virginia Tech
Blacksburg, USA
waris@vt.edu

Ali Anwar
Computer Science and Engineering Department
University of Minnesota Twin Cities
Minneapolis, USA
aanwar@umn.edu

Muhammad Ali Gulzar
Computer Science Department
Virginia Tech
Blacksburg, USA
gulzar@cs.vt.edu

***Abstract*—In Federated Learning (FL), clients independently train local models and share them with a central aggregator to build a global model. Impermissibility to access clients' data and collaborative training make FL appealing for applications with data-privacy concerns, such as medical imaging. However, these FL characteristics pose unprecedented challenges for debugging. When a global model's performance deteriorates, identifying the responsible rounds and clients is a major pain point. Developers resort to trial-and-error debugging with subsets of clients, hoping to increase the global model's accuracy or let future FL rounds retune the model, which are time-consuming and costly.**

We design a systematic fault localization framework, FEDDEBUG, that advances the FL debugging on two novel fronts. First, FEDDEBUG enables interactive debugging of realtime collaborative training in FL by leveraging record and replay techniques to construct a simulation that mirrors live FL. FEDDEBUG's *breakpoint* can help inspect an FL state (round, client, and global model) and move between rounds and clients' models seamlessly, enabling a fine-grained step-by-step inspection. Second, FEDDEBUG automatically identifies the client(s) responsible for lowering the global model's performance without any testing data and labels—both are essential for existing debugging techniques. FEDDEBUG's strengths come from adapting differential testing in conjunction with neuron activations to determine the client(s) deviating from normal behavior. FEDDEBUG achieves 100% accuracy in finding a single faulty client and 90.3% accuracy in finding multiple faulty clients. FEDDEBUG's interactive debugging incurs 1.2% overhead during training, while it localizes a faulty client in only 2.1% of a round's training time. With FEDDEBUG, we bring effective debugging practices to federated learning, improving the quality and productivity of FL application developers.

***Index Terms*—software debugging, federated learning, testing, client, fault localization, neural networks, CNN**

I. INTRODUCTION

Many machine learning models today require private user information for high-quality training. However, users are naturally reluctant to share such data to minimize the risk of privacy violation. To address the above needs, Federated Learning (FL) [37] enables individual participating clients (*e.g.,* smart-home edge devices) to train a machine learning (ML) model on their local data in a privacy-preserving environment and then send the trained model (*e.g.,* the weights of the neural network) to a central aggregator to build a global model. FL trains highly accurate models without ever accessing user data, keeping clients' data privacy intact [22]. With the advent of frameworks like FedML [14] and IBMFL [33], FL is actively used in solving real-world problems [19, 32, 41, 59].

Problems. The support for collaborative yet privacy-preserving training in FL comes at the cost of transparency and comprehension, making debugging prohibitively complicated. For instance, a faulty client can send an inaccurate model to the aggregator either due to noisy labels [17, 27, 28] in the training data or malicious intent to deteriorate the global model's performance [2]–[4, 39]. Finding such a faulty client is challenging due to a large number of unpredictable clients that may not have participated in every round because of a poor network connection or low battery power [45, 52]. The FL training process also spans numerous rounds, significantly increasing the search space for identifying the true culprit round. None of the existing FL frameworks provide debugging and testing support to developers when building FL applications [22]. These developers rely on guesswork and expensive trial-and-error debugging to find a fault-inducing client.

Challenges. FL poses two fundamental challenges when designing a debugging technique. First, in FL deployments, training and testing data are kept private and strictly reside with clients. Access to such data could allow developers to evaluate individual clients' models sent to the aggregator and identify the lowest-performing model as the culprit, similar to traditional ML model testing. Neither test data nor labels are available to an FL application developer and, therefore, existing ML debugging approaches [38, 40, 49] are inapplicable.

Second, due to the unpredictability of clients' participation in a round and the ephemeral nature of their contributions in the global model, reproducing a fault (*i.e.,* faulty client) and then debugging it is not feasible. Traditional breakpoint debugging will pause the entire training process in FL across all clients, causing severe side effects such as data loss as clients may not have persistent storage to store data. Live postmortem or trial-error debugging may lead to a new set of clients for each round based on client availability and quorum, thus making debugging even more ineffective. Considering the above limitations and challenges, we must design a debugging approach that does not rely on clients' data, can debug a live FL application without any interference, and can localize a faulty client precisely.

Contributions. We take inspiration from traditional debuggers, such as `gdb`, and redesign traditional debugging constructs that are tailored to the needs of an FL application developer. Our approach, FEDDEBUG, selectively records an FL application's telemetry data to enable realtime interactive debugging on a simulation that mirrors a live FL application. With FEDDEBUG's *breakpoint*, a developer can spawn a simulation of a live FL application and inspect the current state containing information such as clients' models and their reported metrics (*e.g.,* their training loss or hyperparameters). It also allows a seamless transition between the rounds and clients at a given breakpoint, enabling a fine-grained step-by-step inspection of the application's state. When a developer finds a suspicious state (*e.g.,* multiple clients report high training loss), FEDDEBUG's automated fault localization approach precisely identifies the faulty client(s) without any test data or labels. Once a faulty client is identified, FEDDEBUG's *fix and replay* repairs the global training by retroactively removing the faulty client and resuming the live FL training.

Key Insights. FEDDEBUG leverages several insights to enable systematic FL debugging while preserving clients' privacy. We observe that instead of debugging a live FL application, we can record a set of runtime metrics essential to regenerate a given state in an FL application. Thus, FEDDEBUG performs debugging on a regenerated simulated state equivalent to a live state. To have a measurable impact on the global model, a faulty client's model must behave differently than the regular clients. Every client in an FL application has the same model architecture, so their internal behaviors are comparable. Based on this insight, FEDDEBUG proposes an inference-guided test selection method to select high-quality and diverse test data from a pool of randomly generated input images using Kaiming Initialization [15]. However, an auto-generated data does not include the class label *i.e.,* an oracle. To address the *oracle* problem with such data, FEDDEBUG adapts differential testing to FL domain. It captures differences in the models' execution via neuron activations instead of output labels to identify *diverging* behavior of faulty clients.

Evaluations. We perform large-scale, extensive evaluation of FEDDEBUG on popular models, two large-scale datasets, two well-established FL data distributions, and a real-world fault-injection technique in a total of 68 different FL configurations. We measure FEDDEBUG's fault localizability, debugging time, performance overhead over a vanilla FL framework (IBMFL), and scalability. FEDDEBUG shows remarkable success in identifying faulty clients. It can localize the real-world faulty client with 100% accuracy within 2.1% of a round's training time. When faced with multiple faulty clients, FEDDEBUG retains the high fault localization accuracy of 90.3%. FEDDEBUG's debugging constructs incur an overhead of 48% of the aggregation time to record telemetry data for state regeneration. Surprisingly, this time is only 1.2% of a single round's training time in our experiments. Through our evaluation, we demonstrate that FEDDEBUG effectively conducts interactive debugging and efficiently automates fault localization without incurring high runtime costs. FEDDEBUG augments the IBMFL framework, but its underlying insights can be adapted for other FL frameworks.

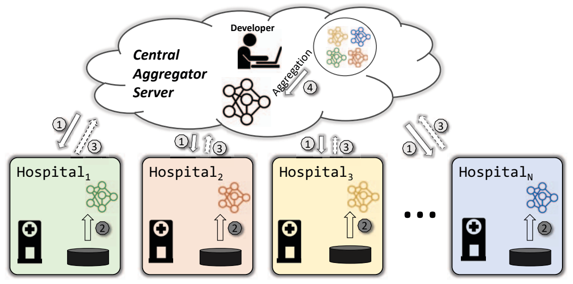

Fig. 1: In a centralized FL architecture, an aggregator sends a global model to clients (step 1). Each client trains the model on local data (step 2) and sends the locally trained model back to the server (step 3). The server aggregates all models to form a new global model (step 4).

We summarize FEDDEBUG's contributions below:

- **Originality:** To the best of our knowledge, FEDDEBUG is the first general-purpose debugging framework for federated learning applications that is not limited by access to clients' data. It addresses open debugging challenges in FL [22].
- **Approach:** Traditional ML trains a single model, whereas FL involves distributed training across hundreds of clients over multiple rounds. Thus, existing ML debugging approaches are inapplicable on FL. FEDDEBUG's novelty lies in observations about FL and the exploitation of insights on reproducibility, inference guided test generation, and differential testing that do not impede performance or violate FL privacy principles.
- **Benchmark:** We evaluate FEDDEBUG in 68 FL configurations derived from well-established datasets, models, varying clients, data distribution, and fault-injections. We package our experiment environment into a public benchmark for future research use.
- **Usefulness:** Our extensive experiments demonstrate that FEDDEBUG successfully locates faulty client(s) without impeding the FL workflow. On a wide range of experiments, FEDDEBUG exhibits robust results against multiple faulty clients, challenging data distributions, and a large number of clients. FEDDEBUG's artifact and the benchmarks used in this work are publicly available at https://doi.org/10.5281/zenodo.7578656.

II. BACKGROUND AND MOTIVATION

A. Federated Learning

In Federated Learning, multiple clients independently train local models on their data and share it with a central server (also called an aggregator) to construct a global model. During this collaborative training, clients' training data never leaves their premises [22]. Figure 1 shows an FL setting where multiple hospitals collaboratively train a global model on their local labeled medical imaging data.

1) In the first step, the aggregator sends copies of the current global model, *i.e.,* the global model weights, and hyperparameters (*e.g.,* learning rate and epochs) to participating clients (Step 1 of Figure 1).
2) Using the global model as initial parameters, each client trains a model on its local data similar to traditional ML training (Step 2 of Figure 1).
3) Once trained, each client sends its local model, in the form of updated weights, back to the aggregator as shown in Step 3 of Figure 1. Additionally, clients share performance metrics such as training loss and quality/quantity of training data with the central aggregator.
4) After receiving model updates, the server aggregates the updated weights from all clients using established model aggregations (also called fusion) techniques such as FedAvg [37] to form a new global model (Step 4).

The four steps are repeated for a fixed number of *rounds* or until the global model meets some convergence criteria, for example, when training loss is close to zero. Note that not every client participates in every round. There are additional variants of federated learning (FL) such as vertical FL [31], FL with differential privacy [50], and personalized FL [44]. Our work mainly focuses on the standard FL [37].

B. Motivating Scenario

Suppose that an FL application developer trains a global neural network model, ResNet [16], on chest X-ray images from hospitals across the country to diagnose respiratory diseases (*e.g.,* Covid-19). The term *developer* refers to a person who writes, deploys, and monitors the FL application at the central server, as shown in Figure 1. Every participating hospital collects X-rays of patients labeled by radiologists and trains a local ResNet model on that data. Hospitals periodically share their locally trained models with a central server. The central server then aggregates these shared models into one global model. After aggregation, the central server sends the updated global model to each hospital to incorporate in local training in the next round, as shown in Figure 1.

The developer observes that multiple hospitals are reporting a high training loss from their preceding training rounds. One plausible reason is that one of the hospitals performed training on noisy data (mislabeled by inexperienced staff [8, 28]) and continuously impacted the global model during aggregation. Thus, when the global model is shared back with the other hospitals, it influences their training.

Challenges of FL Debugging. After noticing an increase in training loss, the developer must investigate the root cause, as misdiagnosis from medical imaging can lead to ill treatment. To debug the FL application at this scale, the developer begins by manually inspecting various collected logs at the central server, including the global model weights, shared local models from hospitals, and the response and training time of each hospital. Due to patient privacy, the hospitals refrain from sharing their labeled training data, which is critical for correctly evaluating the quality of a model and thus essential for localizing the faulty round and model. Even if the developer finds the problematic round, she cannot isolate the hospital(s) responsible for affecting the global model without test data. One option is cross-validating each client's model by requesting that the other clients test the model on their local data. This is prohibited in practice, as it adds computational burden on clients (*e.g.,* edge devices) and can potentially cause data privacy violation. Lastly, statically inspecting hospitals' models does not provide any meaningful information. Without any debugging techniques at her disposal, she resorts to using guesswork to identify the hospital with noisy labels.

FEDDEBUG's Contributions. The developer decides to use FEDDEBUG to investigate the root cause behind high training loss. When enabled, FEDDEBUG allows a developer to set a *breakpoint* at any round or even in the first round to capture the end-to-end training logs. This breakpoint separately invokes a debugging session, a simulation of the original FL service, without stopping the live training. In the debugging session, the developer uses FEDDEBUG's *step-back* and *step-next* constructs to move between rounds, inspecting the global and local models of hospitals. Upon inspecting the training rounds, she finds the specific round, *e.g.,* `round 8`, where the performance starts to deteriorate. This round can be different from the *breakpoint* enabled round, as performance issues can manifest in earlier rounds but surface later. During this inspection, FEDDEBUG also reports the list of hospitals that participated in that round. Next, she invokes FEDDEBUG's fault localization algorithm to precisely identify the hospital responsible for deteriorating the global model performance. After finding the hospital with noisy labels, the developer removes it from the problematic round (*i.e.,* `round 8`) and onwards. FEDDEBUG's *fix and replay* starts retraining from `round 8` to the current round and then replaces the impacted global model with the retrained global model and switches back to the original FL training.

III. FEDDEBUG'S DEBUGGING CONSTRUCTS

The goal of FEDDEBUG is to facilitate an FL application developer in isolating a faulty client responsible for deteriorating the global model performance. Recent studies emphasize the need for debugging techniques in FL applications and the challenges associated with providing debugging support in FL frameworks [22]. To this end, we must overcome the following *major challenges* in designing FEDDEBUG. First, the privacy concerns of FL put restrictions on any client-side interference. Second, the unpredictable and ephemeral nature of clients in FL poses a threat to reproducibility, which is critical for debugging a live system. Third, the distributed nature of FL with hundreds of participating clients makes traditional breakpoint debugging ineffective. Pausing the entire FL application at this scale will be prohibitively expensive. Therefore, traditional debugging approaches, such as `gdb`, are not suitable for the scale and architecture of FL systems.

In FEDDEBUG, we address the above challenges and advance systematic FL application debugging. We enable real-time, interactive debugging on a simulation of the live FL application. To do so, FEDDEBUG continuously collects and

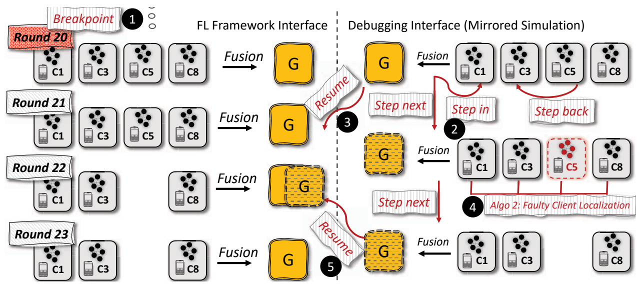

Fig. 2: Using FEDDEBUG, a developer can set a *breakpoint* at `round 20`. When the FL application finishes `round 20`, FEDDEBUG launches a Debugging Interface, reflected on the right. *Step next* (❷) takes the developer to the next step (round or client). *Step-in* increases the granularity of computation, *e.g.,* round to client level. *Resume* (❸) rejoins the current execution status of the FL application if no intrusive actions are taken. At a given round, FEDDEBUG can automatically localize the faulty client (❹) and then resume (❺) upon which the global model will be recomputed without the faulty client. This model will replace the corresponding round's model, and FEDDEBUG will start retraining from that round, `round 22`, in the FL interface.

stores concise telemetry data from a live FL application. Whenever a debugging need arises, the developer can interact with FEDDEBUG's debugging interface, which uses the telemetry data, to regenerate an FL application's state.

A. Selective Telemetry

FEDDEBUG collects critical FL execution metrics to reproduce an FL application's state for the developer to interact with it while investigating the root cause of a problem. Existing FL frameworks are carefully architected to refrain from revealing private data. As a result, most debugging data is private and cannot be investigated.

FEDDEBUG's debugging approach is inspired by replay debugging. As with any other replay debugging approach, it is essential that FEDDEBUG stores the necessary runtime metrics to reproduce an FL application's state if requested by the developer. We design a highly selective FL event telemetry technique that records the concise execution data available at the central aggregator that is vital for generating any prior FL application state. FEDDEBUG is different from traditional replay debugging as it only tracks the information needed to recreate an *observable* event and does not log the information unavailable to the developer in a live application. This design reduces the size of continuously growing telemetry data and minimizes the likelihood of information leakage. FEDDEBUG mainly stores the information available after step 3 of Figure 1 which is clients' models, their reported metrics such as response time, training loss, validation loss, performance metric (*e.g.,* F1 score), hyperparameters (*e.g.,* learning rates, epochs, weight decay), and round ID. Note that the FL application, including client-side training, will continue uninterrupted in the background with FEDDEBUG's telemetry module continuously collecting execution traces.

B. Interactive Replay Debugging

To start the interactive debugging process, a developer can invoke FEDDEBUG's debugging constructs that let the developer leverage the telemetry data to investigate the root cause. Breakpoint debugging is the de-facto method of debugging a program. It pauses the program when the execution reaches it. At that point, a developer can inspect the values assigned to different variables, both local and global, and examine the method stack. Such debugging features are not applicable in FL. The traditional breakpoint will pause the distributed training, resulting in unnecessary idling at the client side. Additionally, since the state of a round is not saved, it is currently impossible for the developer to inspect previous rounds. For instance, a developer may want to debug a latent issue that was introduced by a client five rounds ago but surfaced in the current round when the same client participated in training again.

We make the following observation about FL frameworks. An FL application only reveals aggregator's events to a developer. In contrast, events on the client's side are entirely hidden from the developer except the ones relayed to the aggregator by the client. Building on this observation and the telemetry data captured by FEDDEBUG, our insight is that instead of debugging a system in real-time, we can recreate its observable behavior in a simulated environment, giving an illusion of debugging an FL application in real-time. By doing so, inspections with FEDDEBUG are side-effect free, *i.e.,* FEDDEBUG will not interfere or interrupt the live FL application. Thus, eliminating the need to pause client-side training or halt FL aggregator execution.

Breakpoint. To this end, FEDDEBUG offers *breakpoint* that can help a developer inspect intermediate states of an FL application. FEDDEBUG's breakpoint operates on computation units of *rounds*. Any abnormality in the client-reported metrics, such as training loss, validation loss, response time, and performance metrics (e.g., F1 score) can necessitate the use of breakpoints. FEDDEBUG allows setting a breakpoint at any arbitrary round during live FL. A developer can also set a breakpoint from the start (*i.e.,* `round 0`) to capture end-to-end FL training traces or on a specific round (*e.g.,* `round 20` in Figure 2-❶) to inspect FL training at that round. When the live FL application arrives at a breakpoint, FEDDEBUG spawns a new debugging interface on the aggregator side, as shown in ❶ in Figure 2, while continuing the live FL training in the background.

Step in/Step out. While at a breakpoint in a debugging session, a developer can use *step-in* and *step-out* actions to switch between different granularities of computational units. Traditionally, these two actions are used to go one-level deeper in the stack (*e.g.,* inside a function call) and move one level up in the stack (*e.g.,* outside the function call), respectively. Based on this convention, we define a round as a coarse-grained unit of computation that can be decomposed into a subset of clients participating in that round. Suppose the current breakpoint is at `round 20`. Step-in will take the developer to the clients-level granularity (❷ in Figure 2) where trained models from clients are being aggregated, using a fusion algorithm (*e.g.,* FedAvg [37]). Step-out will take the developer back to the level of rounds, allowing them to inspect the global trained model at a higher level of abstraction and understand its performance across multiple rounds. Inspecting a state at client-level granularity entails evaluating the performance of a partially-aggregated global model. For example, in Figure 2, step-in at ❷ will take the execution between `C1` and `C3`, where the global model has yet to incorporate the local models of clients `C5` and `C8`.

Step Next/Step Back. Similar to step-in/out, *step next* and *step back* help a developer transition from one state to another. For instance, if the breakpoint is at `round 20`, step next will take the execution to `round 21` in the debugging interface, showing information corresponding to that round only. Similarly, if the breakpoint is at client `C5`, step back will take the execution state to a partial global model after aggregating models from clients `C1` and `C3` only (`Step back` in Figure 2).

Resume. Unlike resume in `gdb`, FEDDEBUG's *resume* does not resume any paused execution. Instead, resume gives the illusion to the developer that execution is being continued from where it left off. FEDDEBUG creates this environment by replaying the telemetry data that was captured while the FL application was being inspected using breakpoints, in case the developer does not find any faults in the round under inspection. Once the sequence of events in telemetry catches up with the live execution of the FL application, FEDDEBUG switches to the FL interface and shuts down the debugging interface. This three-step process is nearly indistinguishable from an FL application with FEDDEBUG disabled, giving the impression of debugging a real-time FL application interactively. *Resume* is also illustrated in Figure 2 - ❸.

C. Fix and Replay

When the developer successfully identifies a faulty client in any round, FEDDEBUG offers *Fix and Replay* to allow a developer to roll back the training and provide a retrained global model (the one without a faulty client). We describe the technique to identify a faulty client in Section IV. A faulty client may have a compound effect on the global model, as it may have begun to share its noisy model updates latently several rounds ago, which only later becomes noticeable. In such cases, it is important to rectify the impact of a faulty client's inclusion in prior training rounds by removing its contributions. This requires retraining over multiple rounds, which is not possible as clients may not store the data used in training in the prior rounds. Figure 2-❹ shows the removal of a faulty client (`C5`) in `round 21`. FEDDEBUG recomputes the global model in the debugging interface and then replaces the actual global model in `round 22` with the newly recomputed global model after fix and replay (Figure 2-❺). By default, FEDDEBUG forbids the faulty client from participating in the FL training. However, it is up to the developer to weigh the benefits of including the faulty client in future rounds.

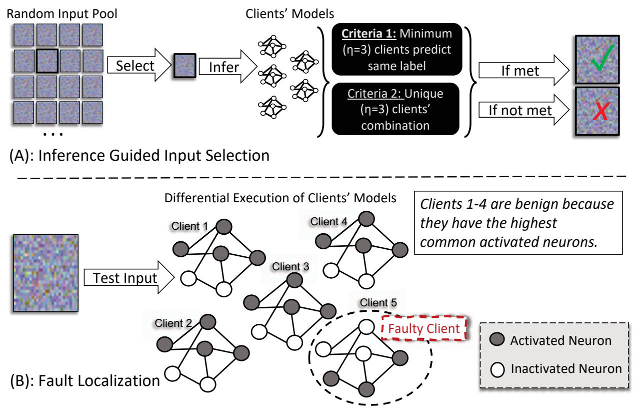
Random Input Pool
Select
Infer
Clients' Models
Criteria 1: Minimum (η=3) clients predict same label
Criteria 2: Unique (η=3) clients' combination
If met
If not met
(A): Inference Guided Input Selection
Differential Execution of Clients' Models
Clients 1-4 are benign because they have the highest common activated neurons.
Test Input
Client 1
Client 2
Client 3
Client 4
Client 5
Faulty Client
Activated Neuron
Inactivated Neuron
(B): Fault Localization

Fig. 3: An overview of FEDDEBUG's fault localization approach. It first selects a random input that invokes diverse model behavior (A). It then applies differential execution on clients' models to localize a faulty client (B).

IV. Faulty Client Localization

Faults in a client's model can arise due to measurement errors, human labeling errors, data poisoning, communication problems, or subjective biases of labellers. For a high-quality global model, it is critical to correctly identify a faulty client and potentially restrict its participation. Manually identifying faulty clients is neither scalable nor effective due to a large number of participating clients in FL and their uninterpretable models. Furthermore, the model parameters (*i.e.,* weights) do not provide any meaningful debugging information. To automate faulty client localization, we must define a feedback mechanism to guide our search for faulty clients efficiently. Automated debugging tools [26, 55] for regular software address this problem by relying on multiple test *inputs* and a test *oracle*. For example, unit tests can guide the search toward concise input leading to incorrect program output [55]. In FL, the inputs and oracle translate into diverse test data and the corresponding accurate labels, both of which are unavailable to the developer at the central server.

FEDDEBUG addresses the challenges of automated fault localization with a two-pronged approach. First, it generates a pool of random test inputs and applies a novel inference-guided test input selection to construct a suite of test inputs, as shown in Figure 3-A. Since the test inputs are generated autonomously and are not accompanied by ground truth labels, metrics such as F1 score or accuracy cannot be used as oracle feedback to identify a faulty client. Instead, FEDDEBUG performs differential testing of clients' models to measure similarities and differences among models' behaviors on selected inputs (Figure 3-B). FEDDEBUG fingerprints a neural network behavior on an input by profiling the internal neurons' contributions towards a model prediction. Subsequently, FEDDEBUG accurately recognizes a client as faulty if its behavior deviates from the norm, which is the majority of the clients' behavior. Our insight is that a faulty client's model

Algorithm 1: Inference-Guided Test Input Selection

```
Input: $shape$: dimension of the random input to be generated.
Input: $\kappa$: number of inputs to be generated.
Input: $\eta$: minimum number of clients for the same prediction.
Output: $\overline{X}$: a list containing auto-generated test inputs.

$rand\_inputs = lazilyGenerateRandInputs(shape)$
$\overline{X} = list()$ // a list for inference guided test inputs
$seen\_clients\_sequences = list()$
while $length(\overline{X}) < \kappa$ do
    $r\_input = pop(rand\_inputs)$
    $clients\_preds = getPredictions(clients, r\_input)$
    for $label \in class\_labels$ do
        $clients\_seq = samePredClients(clients\_preds, label)$
        if $clients\_seq \notin seen\_clients\_sequences$ and $length(clients\_seq) \geq \eta$ then
            $seen\_sequences.append(clients\_seq)$
            $\overline{X}.append(r\_input)$ // valid test input
            break
    if $length(rand\_inputs) < 1$ then
        $rand\_inputs = lazilyGenerateRandInputs(shape)$
return $\overline{X}$
```

Algorithm 2: Faulty Client Localization using Differential Testing

```
Input: $clients$: a list of clients participated in the given FL round.
Input: $x$: a random input belongs to $\overline{X}$.
Input: $na\_t$: a threshold to profile neuron activations.
Output: $faulty\_client$: the faulty client who has abnormal behavior.

$all\_clients\_combinations = nChooseK(clients, 1)$
$benign\_clients = set()$
$max\_common\_activations = -1$
for $t\_clients \in all\_clients\_combinations$ do
    $neuron\_ids = ActivatedNeurons(t\_clients, x, na\_t)$
    $t\_clients\_common\_neurons = intersection(neuron\_ids)$
    $temp\_n = length(t\_clients\_common\_neurons)$
    if $temp\_n > max\_common\_activations$ then
        $max\_common\_activations = temp\_n$
        $benign\_clients = t\_clients$
$faulty\_client = clients - benign\_clients$
return $faulty\_client$
```

will show a noticeable difference in its internal neuron values compared to benign clients' models, based on the principle that faulty executions are intrinsically different from correct ones. The same principle is behind popular fault localization techniques, such as spectra-based fault localization [21] and delta debugging [55].

Inference-Guided Test Input Selection. As shown in Figure 3-A, FEDDEBUG first lazily generates a pool of random test inputs using Kaiming Initialization [15]. For example, if the clients' models are trained on 32x32 images within the RGB scale, then FEDDEBUG randomly creates a pool of synthetic inputs with the same size and format (*i.e.,* random images of size 32x32 in RGB scale). It then automatically selects only those inputs that lead to a consensus on predictions among a *unique* subset of clients. FEDDEBUG selects up to κ test inputs (default is $\kappa = 10$) among the pool of 1000 random inputs. The goal is to minimize any overlapping behavior between clients while inferring unique class labels on selected test inputs. This is similar to achieving maximum code coverage in regular software with minimum tests. Algorithm 1 selects a test input (line 5) if at least ($\eta \geq 5$) clients predict the same label and that subset of clients has not been seen in a previously selected input (lines 6-11). On the next random input, if the previously observed subset of clients (*i.e.,* $clients\_seq \in seen\_clients\_sequences$) predict the same class label, we discard this input. If a unique combination of clients predicts an unseen label, we include the input in the test suite. This process is repeated until we collect a user-defined, κ, number of test inputs.

Differential Execution of Clients Models. In the absence of correct labels of generated test inputs, FEDDEBUG adapts differential testing to find behavioral differences and similarities among clients' models, as shown in Figure 3-B. FEDDEBUG profiles the contributions of individual neurons during model inference on an input and uses these neurons activations to identify models with common behavior. Note that clients' models in FL are comparable due to having a similar architecture. Algorithm 2 describes the faulty client localization process. For a selected test input, FEDDEBUG exhaustively iterates all possible combinations of potentially non-faulty clients (*i.e.,* $\binom{n}{1}$ combinations). For each combination, Algorithm 2 performs model inference on the test input and captures its neuron profiles. FEDDEBUG aims to find one combination of clients that has the highest overlap in behavior, representing the true $n-1$ benign clients and consequently isolating the precise faulty client. This is a lightweight process due to the negligible model inference time and the iterations' linear time ($O(n)$) complexity.

Our insight is that among all possible combinations of clients, only one represents true benign clients' subset. The remaining combinations contain the faulty client with abnormal neuron activations, reducing the model behavior overlap within that set. In summary, at a given ill-performing round in FL, FEDDEBUG takes in all participating clients' models as the only input. It automatically generates test inputs and employs differential testing on clients' models to monitor abnormal behavior to precisely identify a faulty client.

V. EVALUATION

We evaluate FEDDEBUG on (1) runtime performance overhead, (2) debugging time, (3) fault localizability, and (4) scalability. Our evaluation aims to answer the following research questions:

- **RQ1.** What impact does FEDDEBUG have on the baseline FL framework's performance?
- **RQ2.** How accurate is FEDDEBUG in identifying a faulty client?
- **RQ3.** Can FEDDEBUG identify multiple faulty clients?
- **RQ4.** Can FEDDEBUG scale to large number of clients and find a faulty client efficiently?

Datasets, Models, and FL Framework. We evaluate FEDDEBUG on CIFAR-10 and FEMNIST datasets. Both are considered as gold standard to evaluate FL experimental settings [9, 30]. FEMNIST is a modified version of MNIST presented in the FL LEAF Benchmark [7] and the Non-IID Bench [30]. The FEMNIST dataset includes more than 340K training and 40K testing grayscale images, each with a resolution of 28x28 pixels, representing ten distinct class labels. CIFAR-10 contains 50K training 32x32 RGB images that span ten different classes and 10K instances for testing. We adopt popular CNN models, namely ResNet [16], VGG [43], and DenseNet [18] architectures. We set the learning rate between 0.0001 and 0.001, the number of epochs between 10 and 25, the batch size from 512 to 2048, and the weight decay to 0.0001. We realize FEDDEBUG's design in the IBMFL library [33] due to its ease of use, open documentation, and publicly available codebase. These techniques should be equally applicable to other FL frameworks.

Evaluation Environment Specifications. We run our experiments on an AMD 16-core processor with 128 GB RAM and an NVIDIA Tesla T4 GPU. To measure the performance of FEDDEBUG in terms of runtime and debugging overhead, we simulate IBMFL framework deployment on a MacBook Pro with a Quad-core Intel Core i5 processor and 16 GB RAM.

Federated Learning Experimental Settings. Prior FL literature [7, 30] establishes two data distribution strategies among FL clients: IID (independent and identically distributed data) and Non-IID (non-independent and identically distributed data). For Non-IID, we use the quantity base imbalance [30] where clients have an unequal quantity of data, and the class distribution is random. In IID, the clients receive the same quantity of data. None of the clients share the same data points in both settings. We simulate FL with a varying number of clients, ranging from 10 to 400 clients, in each FL training round. In practice, even with millions of clients, only a subset (in the order of hundreds) is selected in a round. Therefore, our experiment settings are representative of real-world FL deployments [1, 5, 24, 30, 37, 48].

Fault Injection. Since there is no existing FL benchmark with faulty clients, FEDDEBUG adopts a standard noisy labels approach from prior machine learning literature to inject a faulty client in our experiments [10, 17, 20, 29, 53]. Similar to prior work [11, 27, 36], we arbitrarily add noise by changing training data labels (*e.g.,* changing label "bird" to "cat"). When such a client's model is merged with the global model, the global model's performance (*e.g.,* accuracy) deteriorates. We define different strengths of a fault with a *noise rate* that controls the number of labels modified in a faulty client. Noise rate is defined as a ratio between changed labels and original labels (*changed-labels/original-labels*).

Figure 4 shows the impact of different noise rates on the global model's accuracy, with one faulty client and nine benign clients. Low noise rates, ranging from 0.2 to 0.7, barely affect the global model performance. With a 0.7 noise rate, the accuracy is lowered by 4.8% and 5.5% in CIFAR-10 and FEMNIST, respectively. A noise rate of 0.9 incurs a 16.2% and 9.9% reduction in the global model accuracy in both settings. Thus, to have a measurable impact on the global model's performance, we select a noise rate of one for a faulty client. Still, we perform sensitivity analysis in Section V-B (Figure 7) by measuring the impact of varying noise rates on FEDDEBUG's fault localizability.

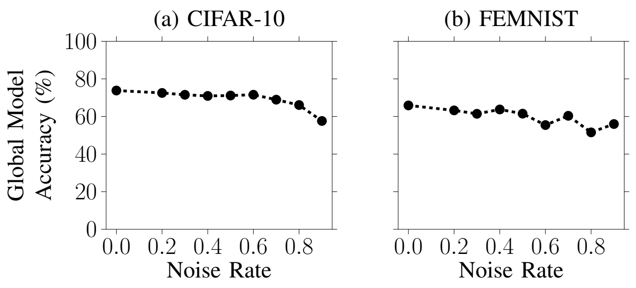

Fig. 4: Global model (ResNet-34) prediction accuracy in the presence of a faulty client with different noise rates. Lower noise rates hardly degrade global model performance.

Neuron Activation Threshold. We adopt the method from Harel-Canada et al. [13] to profile neuron activations. We empirically find 0.003 as the optimal value for the default activation threshold (see Section V-C). A neuron is considered active when its value crosses this threshold.

Faulty Client Localization Accuracy. We calculate faulty client localization accuracy as the ratio between (a) the number of test inputs on which faulty clients are correctly identified and (b) the total number of test inputs. For instance, if FEDDEBUG identifies the correct set of faulty clients on four out of ten test inputs generated by Algorithm 1, we report 40% fault localization accuracy.

A. FedDebug's Performance

Capturing telemetry data in realtime may slow down the performance of an FL application's aggregator. In this subsection, we present the evaluation results of FEDDEBUG's runtime overhead and the fault localization time. These experiment settings employ ResNet-18 with CIFAR-10.

Runtime Overhead (RQ1). To evaluate the impact on the FL application's performance, we measure the slowdown in the running time that FEDDEBUG incurs. We compare the cumulative processing time of the vanilla IBMFL's aggregator (baseline) against that of the FEDDEBUG-enabled aggregator on a variety of client combinations, ranging from 5 clients to 100 clients. The aggregation time varies with the model's architecture and the number of clients participating in a round, but it is completely independent of the models' quality. Therefore, we create up to 100 pre-trained ResNet-18 models and perform the aggregation.

Figure 5 compares the baseline's aggregation time with the FEDDEBUG enabled aggregation time. The X-axis represents the number of clients ranging from 5 to 100 clients, and the Y-axis represents the average time across two FL rounds. For instance, with 30 clients, FEDDEBUG takes 3.9 seconds compared to the 2.5 seconds for the baseline to aggregate 30 trained models into a global model. Overall, FEDDEBUG takes approximately 48% additional aggregation time across all experiments. However, in an end-to-end round, the training phase on the clients' end occupies the majority (up to 97.8% in

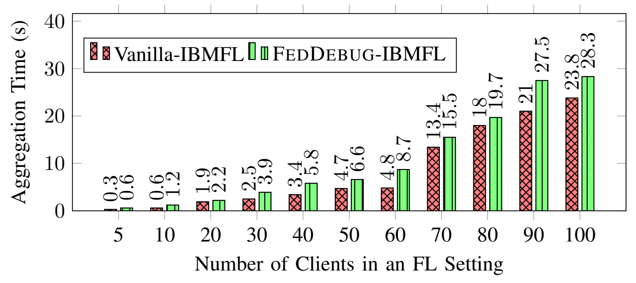

Fig. 5: FedDebug's runtime overhead as a comparison between vanilla FL framework's aggregation time with FedDebug enabled FL aggregation.

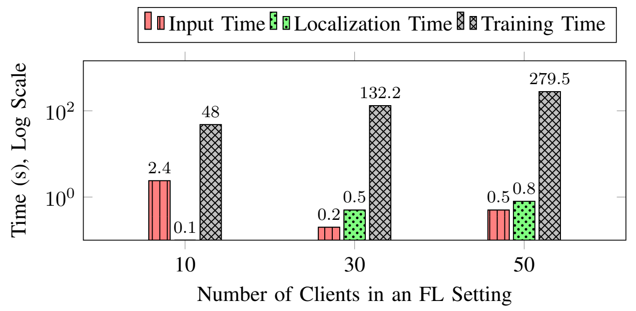

Fig. 6: FedDebug's debugging time contains input generation time and faulty client detection time and is compared against a round's training time.

our experiments) of the round's time. Compared to the training time of a round, the aggregation time is almost negligible, as low as 1.2% in our experiments.

Summary: Considering the training and aggregation time of each FL round, FedDebug's runtime overhead is a very small fraction, 1.2%, of the training time. Hence, capturing telemetry data for replay debugging does not impede the FL application's runtime performance.

Debugging Time (RQ1). To assess the localizability of FedDebug, we design experiments to measure FedDebug's *debugging time*, the time it takes to localize a faulty client. We then compare this time with the training time of that round. Since there is no comparable approach to localize a faulty client, we use training time as a baseline to provide a meaningful scale for the cost of debugging.

Figure 6 shows the results of these experiments. The X-axis represents the number of clients, and the Y-axis shows the debugging time in seconds on a logarithmic scale. For 30 clients, FedDebug's input generation and selection takes 0.2 seconds to find high-quality test input, and its fault localization takes approximately 0.5 seconds to localize a faulty client. In a ten clients setting, input selection takes longer due to constraint $\eta = 4$ for criteria 1 in Figure 3. $\eta = 4$ means that at least four previously unseen clients should predict the same label on newly selected test input.

TABLE I: FedDebug's debugging time and accuracy when localizing a faulty client in 36 different FL settings with 100 test inputs.

Clients	Dataset	Architecture	Accuracy % (IID)	Accuracy % (Non-IID)	Avg. Input Time (s)	Avg. Localization Time (s)
10	CIFAR10	DenseNet-121	100	100	2.41	0.44
10	CIFAR10	ResNet-50	100	100	2.40	0.22
10	CIFAR10	VGG-16	100	100	2.40	0.21
30	CIFAR10	DenseNet-121	100	100	2.42	1.29
30	CIFAR10	ResNet-50	100	100	1.18	0.70
30	CIFAR10	VGG-16	100	100	2.41	0.47
50	CIFAR10	DenseNet-121	100	100	2.42	3.26
50	CIFAR10	ResNet-50	100	100	1.37	1.24
50	CIFAR10	VGG-16	100	100	2.43	0.91
10	FEMNIST	DenseNet-121	100	100	2.40	0.47
10	FEMNIST	ResNet-50	100	100	2.40	0.25
10	FEMNIST	VGG-16	100	100	2.40	0.18
30	FEMNIST	DenseNet-121	100	100	2.41	1.37
30	FEMNIST	ResNet-50	100	100	0.91	0.68
30	FEMNIST	VGG-16	100	100	2.41	0.55
50	FEMNIST	DenseNet-121	100	100	2.24	2.44
50	FEMNIST	ResNet-50	100	100	1.42	1.24
50	FEMNIST	VGG-16	100	100	2.40	1.25

Overall, our results show an increasing debugging time when the number of clients increases, which is expected as increasing the number of clients increases the search space. Note that the debugging time is still in the order of seconds, even for 50 clients. This is because 1) for n clients, the search space has at most n possible combinations of potentially benign n−1 clients, representing linear complexity, and 2) on a given input, FedDebug only profiles neuron activations once while iterating over the n combinations.

Summary: On average, FedDebug can efficiently identify a faulty client in 2.1% of the total training time of a round.

B. Localization of Faulty Client(s)

To answer RQ2, we measure how accurate FedDebug is in localizing a faulty client. We inject a faulty client that is representative of a real-world scenario and can cause a measurable change in the global model's performance. By varying the number of clients, datasets, models, and data distributions (IID and Non-IID), we create 36 different FL configurations for FedDebug's evaluation.

Column 4 and 5 of Table I show the accuracy of FedDebug in the IID and Non-IID settings, respectively. We repeat each experiment on 100 generated test inputs and take the average of each metric to generalize the results. FedDebug correctly identifies a faulty client with 100% accuracy in both IID and Non-IID settings.

Varying Noise Rate. Figure 4 shows the impact of different noise rates on the global model prediction accuracy. We observe that a faulty client has a measurable impact on the global model with a noise rate of > 0.8. The global model's accuracy merely drops from 73.8% to 71.1% when the faulty client has a 0.6 noise rate, and drops to 57% when the noise rate is close to one. FedDebug localizes faulty client(s) with low noise rates, showing its robustness. Figure 7 shows the evaluations on varying noise rates in 10 clients FL settings with ResNet and DenseNet architectures. The X-axis shows

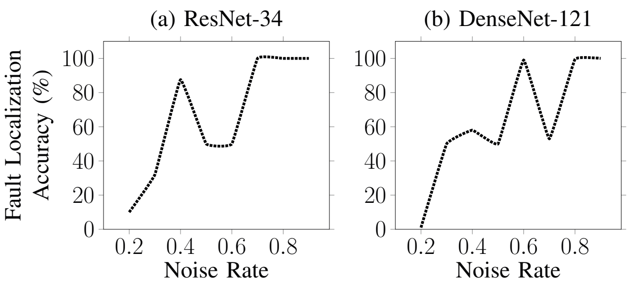

Fig. 7: FEDDEBUG localization performance when a faulty client has varying fault strength (*i.e.,* low noise rate).

the faulty client's noise rate, and the Y-axis represents the average fault localization accuracy on the CIFAR-10 and FEMNIST datasets. The results, as seen in Figure 7, indicate that FEDDEBUG has the capability to identify low noise faults—it successfully localizes a faulty client with 0.4 noise rate with approximately 58% and 87.5% accuracy in DenseNet and ResNet settings, respectively.

Summary: FEDDEBUG achieves 100% fault localization accuracy on average on a total of 3600 test inputs when the faulty client significantly deteriorates the global model performance in both IID and Non-IID settings. It also accurately localizes a faulty client with low noise rates.

Detecting Multiple Faulty Clients (RQ3). We evaluate FEDDEBUG's ability to identify multiple faulty clients. To this end, we inject up to seven faulty clients in the following experiment settings. We train ResNet-50 and DenseNet-121 on the CIFAR-10 and FEMNIST datasets in 30 and 50 clients FL settings. Each setting is evaluated on 10 test inputs. By default, FEDDEBUG's fault localization technique finds a single faulty client. We apply FEDDEBUG in an iterative manner to find multiple faulty clients by removing one faulty client on each iteration, similar to traditional bug repair process, where one bug is fixed first before the next one is investigated.

Table II presents the results of finding multiple faulty clients in 32 FL configurations. For instance, when 7 out of 30 clients are faulty and the model is ResNet-50, FEDDEBUG finds all seven faulty clients with 100% accuracy on CIFAR-10 and 97.1% accuracy on FEMNIST. Compared to ResNet, FEDDEBUG performs relatively better with DenseNet. This behavior is expected because, compared to ResNet, DenseNet learns better features due to dense concatenation among its layers, resulting in better performance [58]. Thus, FEDDEBUG performs well in localizing multiple faults with DenseNet with an average accuracy of 99.7% on both datasets compared to ResNet's 80.8%.

Table II also reveals that, generally, FEDDEBUG's localization performance is positively correlated to the number of training data points per client. Large, high-quality training data promotes better feature learning among neurons and, thus, yields better performance. Since the number of data points in FEMNIST (340K) is large compared to CIFAR-10 (40K), clients in the FEMNIST settings have significantly

TABLE II: FEDDEBUG's fault localization in 32 FL configurations with multiple faulty clients, ranging from two to seven.

Faulty Clients	Total Clients	Architecture	Accuracy % (CIFAR-10)	Accuracy % (FEMNIST)
2	30	ResNet-50	100	100
3	30	ResNet-50	100	100
5	30	ResNet-50	100	98
7	30	ResNet-50	100	97.1
2	30	DenseNet-121	100	100
3	30	DenseNet-121	100	100
5	30	DenseNet-121	100	100
7	30	DenseNet-121	100	100
2	50	ResNet-50	50	80
3	50	ResNet-50	66.7	66.7
5	50	ResNet-50	54	60
7	50	ResNet-50	57.1	62.9
2	50	DenseNet-121	100	100
3	50	DenseNet-121	100	100
5	50	DenseNet-121	100	100
7	50	DenseNet-121	100	95.7

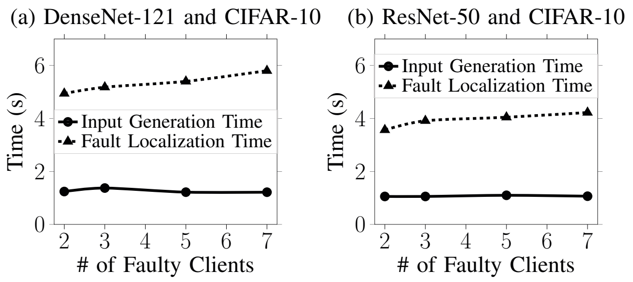

Fig. 8: FEDDEBUG finds multiple faulty clients in a linear time. Total clients are 50 in each graph.

larger training data than clients in the CIFAR-10 settings. As a result, FEDDEBUG average localization accuracy is 78.5% in the ResNet-CIFAR experiment, while it has 83.1% localization accuracy in the ResNet-FEMNIST experiment. FEDDEBUG finds multiple faults with linear time complexity, as shown in Figure 8 with 50 clients. The input generation time is almost constant, as the number of clients is fixed. However, the localization time increases as we increase the number of faults from 2 to 7. For instance, it localizes two faulty clients in 3.6 seconds and five faulty clients in 4 seconds.

Scalability (RQ4). Our findings also show that FEDDEBUG scales to larger datasets and an increasing number of clients in FL. Figure 9 summarizes the impact on FEDDEBUG's ability to identify a faulty client when the number of clients changes from 25 to 400 and the training data size per client changes. We perform this experiment with two faulty clients in the FEMNIST-DenseNet configuration. Figure 9-(a) verifies that FEDDEBUG's fault localization accuracy only reduces to 75% even when the number of clients increases to 400. FEDDEBUG's debugging time increases linearly as the number of clients increases, consistent with the scale-up properties of general distributed systems, as shown in Figure 9-(b). When the number of clients increases, less data is used to train a client's model, which may reduce the accuracy of clients' models. Figure 9-(c) also shows that FEDDEBUG's

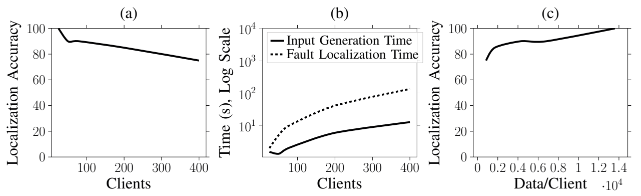

Fig. 9: FEDDEBUG retains scalability on a large number of clients.

fault localizability also increases when the number of data points per client increases, and it is also robust against low performing client models. For instance, when the number of data points increases from 850 to 1700, FEDDEBUG's localization accuracy also changes from 75% to 85%, respectively.

Summary: Our experiment results provide concrete evidence that FEDDEBUG preserves scalability properties both in terms of time overhead and in the presence of multiple faults. It successfully identifies multiple faulty clients in 32 different FL configurations with an average accuracy of 90.3%.

C. Neuron Activation Threshold

There is no standard threshold of neuron activations [40] and prior work uses experiential value for different use cases [13]. We evaluate the impact of different activation thresholds on FEDDEBUG's faulty client localizability. We take 30 clients including five faulty clients, and train ResNet-50 and DenseNet-121 on both the CIFAR-10 and FEMNIST datasets. We repeat each experiment on 10 different inputs generated by Algorithm 1.

Figure 10 shows the result of these experiments. The X-axis represents the neuron activation thresholds, ranging from 0 to 0.9. The Y-axis shows the FEDDEBUG's localization accuracy in a given experiment setting. For instance, at the 0.003 threshold, the average localization accuracy across four settings is 100%. On the other hand, at 0.5 threshold, the average accuracy decreases significantly to 73.5% across these configurations. Specifically, for DenseNet-121 and FEMNIST experiment in Figure 10-(d), the localization drops to 64% at the 0.5 neuron activation threshold. We observe that FEDDEBUG performs better at lower thresholds (< 0.01) across different models and datasets. This behavior is expected because lower thresholds increase the sensitivity of FEDDEBUG's localization approach. It starts monitoring most of the neurons' compared to a higher threshold, where FEDDEBUG profiles only a few neurons crossing the threshold.

D. Threats to Validity

To alleviate threats to external validity, we use established state-of-the-art FL experimental models (ResNet-18, ResNet-34, ResNet-50, DenseNet-121, and VGG-16), two standardized datasets from FL benchmarks, two real-world data distributions, and an industrial scale FL framework. Similarly, we remove bias in fault injection using standard noisy labels technique from the ML literature, to make a fault reflective

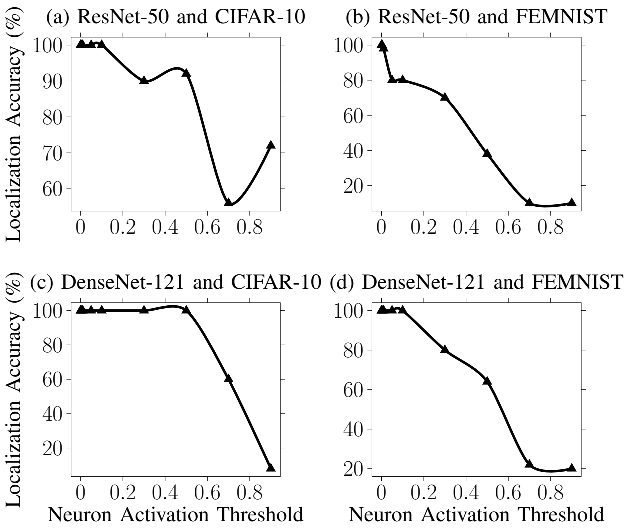

Fig. 10: FEDDEBUG performance at neuron activation threshold on 30 clients, including five faulty clients.

of real-world scenarios. We also experiment with varying noise rates for better evaluations, transparency, and fairness. Another source of external threats to validity is randomness in FEDDEBUG's input selection method. We minimize such randomness by evaluating each configuration on at least 10 and 100 test inputs and reporting the average results.

VI. RELATED WORK

Debugging ML models has been extensively explored in recent works [6, 12, 38, 40, 47, 49, 51]. The primary objectives of these approaches are interpretability, generating new test cases by carefully perturbing the real-world training inputs to improve performance and to find bugs and corner cases in the given model. These approaches require access to the training and testing data, and some are limited to testing a single neural network; hence, such approaches cannot be directly imported into FL. Lack of access to client data and resources in FL settings makes testing and debugging FL more challenging. If applied to FL, these testing approaches will find every client's model defective. Clients' models are architecturally similar but trained on local clients' data, and thus their models are semantically different from each other. Identifying defects in an isolated model is not practical either. Every client's model has weaknesses that will surface on carefully selected test data. FEDDEBUG overcomes these problems by focusing on the commonality of models instead of differences.

Most relevant work to FEDDEBUG primarily focuses on finding clients' contributions to a global model without exposing the private data to a central server [56]. In practice, individual clients report information about training, such as dataset size and performance metrics, to the central aggregator [23, 25, 42, 54, 57]. Existing approaches use prior information *e.g.,* previous task performance and data quality obtained via third-party services, to evaluate clients' models [46]. Other approaches recommend cross-validating clients' models on another client's local dataset [35]. Another alternate is maintaining a validation dataset at the central server to evaluate clients' models [8, 34]. A major limitation of the above FL-related approaches is that the aggregator server depends entirely on the client's reported information or test data to evaluate clients' models. The aggregator also assumes that all clients are trustworthy about their performance in these approaches, which attracts adversarial clients like the ones in targeted poisoning attacks [39]. Cross-validation is also prohibited due to limited computing resources for edge devices such as smart home sensors. FEDDEBUG overcomes the limitations of debugging faulty clients with interactive and automated approaches that preserve privacy.

VII. CONCLUSION

Federated learning promotes collaborative model training across millions of clients—the type of learning that was previously impossible due to privacy concerns related to user data. However, FL poses unprecedented challenges in debugging a faulty client responsible for deterring global training. With minimal information about the training process and non-existent debugging techniques, such issues are often left untreated. FEDDEBUG enables interactive and automated fault localization in FL. It adapts conventional debugging practices in FL with its *breakpoint* and *fix and replay* feature. It offers a novel differential testing technique to automatically identify the precise faulty clients. We demonstrate that FEDDEBUG identifies a faulty client with 100% accuracy within 2.1% of a round's training time, advocating for FEDDEBUG's efficacy and efficiency. With FEDDEBUG, we pave the way for advanced software debugging techniques to be adapted in the emerging area of federated learning and the broader community of machine learning practitioners.

ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation under grant number CCF-2106420. We would like to thank the ICSE reviewers for their constructive feedback.

REFERENCES

[1] Dmitrii Avdiukhin and Shiva Kasiviswanathan. Federated learning under arbitrary communication patterns. In *International Conference on Machine Learning*, pages 425–435. PMLR, 2021.

[2] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *International Conference on Artificial Intelligence and Statistics*, pages 2938–2948. PMLR, 2020.

[3] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. In *International Conference on Machine Learning*, pages 634–643. PMLR, 2019.

[4] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1467–1474, 2012.

[5] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, et al. Towards federated learning at scale: System design. *Proceedings of machine learning and systems*, 1:374–388, 2019.

[6] Houssem Ben Braiek and Foutse Khomh. Deepevolution: A search-based testing approach for deep neural networks. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 454–458. IEEE, 2019.

[7] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*, 2018.

[8] Yiqiang Chen, Xiaodong Yang, Xin Qin, Han Yu, Biao Chen, and Zhiqi Shen. Focus: Dealing with label quality disparity in federated learning. *arXiv preprint arXiv:2001.11359*, 2020.

[9] Liam Collins, Hamed Hassani, Aryan Mokhtari, and Sanjay Shakkottai. Exploiting shared representations for personalized federated learning. In *International Conference on Machine Learning*, pages 2089–2099. PMLR, 2021.

[10] Benoît Frénay and Michel Verleysen. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5):845–869, 2013.

[11] Aritra Ghosh, Himanshu Kumar, and P. S. Sastry. Robust loss functions under label noise for deep neural networks. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 1919–1925. AAAI Press, 2017.

[12] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743, 2018.

[13] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.

[14] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. FedML: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*, 2020.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Dan Hendrycks, Mantas Mazeika, Duncan Wilson, and Kevin Gimpel. Using trusted data to train deep networks on labels corrupted by severe noise. *Advances in neural information processing systems*, 31, 2018.

[18] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[19] Ji Chu Jiang, Burak Kantarci, Sema Oktug, and Tolga Soyata. Federated learning in smart city sensing: Challenges and opportunities. *Sensors*, 20(21):6230, 2020.

[20] Lu Jiang, Di Huang, Mason Liu, and Weilong Yang. Beyond synthetic noise: Deep learning on controlled noisy labels. In *International Conference on Machine Learning*, pages 4804–4815. PMLR, 2020.

[21] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[22] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1-2):1–210, 2021.

[23] Jiawen Kang, Zehui Xiong, Dusit Niyato, Han Yu, Ying-Chang Liang, and Dong In Kim. Incentive design for efficient federated learning in mobile networks: A contract theory approach. In *2019 IEEE VTS Asia Pacific Wireless Communications Symposium (APWCS)*, pages 1–5. IEEE, 2019.

[24] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtarik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[25] Tra Huong Thi Le, Nguyen H Tran, Yan Kyaw Tun, Minh NH Nguyen, Shashi Raj Pandey, Zhu Han, and Choong Seon Hong. An incentive mechanism for federated learning in wireless cellular networks: An auction approach. *IEEE Transactions on Wireless Communications*, 20(8):4874–4887, 2021.

[26] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[27] Junnan Li, Richard Socher, and Steven C.H. Hoi. Dividemix: Learning with noisy labels as semi-supervised learning. In *International Conference on Learning Representations*, 2020.

[28] Junnan Li, Yongkang Wong, Qi Zhao, and Mohan S Kankanhalli. Learning to learn from noisy labeled data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5051–5059, 2019.

[29] Junnan Li, Caiming Xiong, and Steven CH Hoi. Learning from noisy data with robust representation learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9485–9494, 2021.

[30] Qinbin Li, Yiqun Diao, Quan Chen, and Bingsheng He. Federated learning on non-iid data silos: An experimental study. In *IEEE International Conference on Data Engineering*, 2022.

[31] Yang Liu, Yan Kang, Liping Li, Xinwei Zhang, Yong Cheng, Tianjian Chen, Mingyi Hong, and Qiang Yang. A communication efficient vertical federated learning framework. *Scanning Electron Microsc Meet at*, 2019.

[32] Guodong Long, Yue Tan, Jing Jiang, and Chengqi Zhang. Federated learning for open banking. In *Federated learning*, pages 240–254. Springer, 2020.

[33] Heiko Ludwig, Nathalie Baracaldo, Gegi Thomas, Yi Zhou, Ali Anwar, Shashank Rajamoni, Yuya Ong, Jayaram Radhakrishnan, Ashish Verma, Mathieu Sinn, et al. IBM Federated Learning: an Enterprise Framework White Paper V0. 1. *arXiv preprint arXiv:2007.10987*, 2020.

[34] Lingjuan Lyu, Xinyi Xu, Qian Wang, and Han Yu. Collaborative fairness in federated learning. In *Federated Learning*, pages 189–204. Springer, 2020.

[35] Lingjuan Lyu, Jiangshan Yu, Karthik Nandakumar, Yitong Li, Xingjun Ma, Jiong Jin, Han Yu, and Kee Siong Ng. Towards fair and privacy-preserving federated deep models. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2524–2541, 2020.

[36] Xingjun Ma, Yisen Wang, Michael E Houle, Shuo Zhou, Sarah Erfani, Shutao Xia, Sudanthi Wijewickrema, and James Bailey. Dimensionality-driven learning with noisy labels. In *International Conference on Machine Learning*, pages 3355–3364. PMLR, 2018.

[37] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[38] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[39] Mustafa Safa Ozdayi, Murat Kantarcioglu, and Yulia R Gel. Defending against backdoors in federated learning with robust learning rate. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9268–9276, 2021.

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

[41] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletari, Holger R Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N Galtier, Bennett A Landman, Klaus Maier-Hein, et al. The future of digital health with federated learning. *NPJ digital medicine*, 3(1):1–7, 2020.

[42] Yunus Sarikaya and Ozgur Ercetin. Motivating workers in federated learning: A stackelberg game perspective. *IEEE Networking Letters*, 2(1):23–27, 2019.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[44] Canh T Dinh, Nguyen Tran, and Josh Nguyen. Personalized federated learning with moreau envelopes. *Advances in Neural Information Processing Systems*, 33:21394–21405, 2020.

[45] Shunpu Tang, Wenqi Zhou, Lunyuan Chen, Lijia Lai, Junjuan Xia, and Liseng Fan. Battery-constrained federated edge learning in uav-enabled iot for b5g/6g networks. *Physical Communication*, 47:101381, 2021.

[46] Muhammad Habib ur Rehman, Ahmed Mukhtar Dirir, Khaled Salah, Ernesto Damiani, and Davor Svetinovic. Trustfed: a framework for fair and trustworthy cross-device federated learning in iiot. *IEEE Transactions on Industrial Informatics*, 17(12):8485–8494, 2021.

[47] Muhammad Usman, Yannic Noller, Corina S Păsăreanu, Youcheng Sun, and Divya Gopinath. Neurospf: A tool for the symbolic analysis of neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 25–28. IEEE, 2021.

[48] Jianyu Wang, Qinghua Liu, Hao Liang, Gauri Joshi, and H Vincent Poor. Tackling the objective inconsistency problem in heterogeneous federated optimization. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pages 7611–7623, 2020.

[49] Mohammad Wardat, Wei Le, and Hridesh Rajan. Deeplocalize: fault localization for deep neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 251–262. IEEE, 2021.

[50] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. Federated learning with differential privacy: Algorithms and performance analysis. *IEEE Transactions on Information Forensics and Security*, 15:3454–3469, 2020.

[51] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. Diffchaser: Detecting disagreements for deep neural networks. In *IJCAI*, pages 5772–5778, 2019.

[52] Zichen Xu, Li Li, and Wenting Zou. Exploring federated learning on battery-powered devices. In *Proceedings of the ACM Turing Celebration Conference-China*, pages 1–6, 2019.

[53] Jiangchao Yao, Jiajie Wang, Ivor W Tsang, Ya Zhang, Jun Sun, Chengqi Zhang, and Rui Zhang. Deep learning from noisy image labels with quality embedding. *IEEE Transactions on Image Processing*, 28(4):1909–1922, 2018.

[54] Dongdong Ye, Rong Yu, Miao Pan, and Zhu Han. Federated learning in vehicular edge computing: A selective model aggregation approach. *IEEE Access*, 8:23920–23935, 2020.

[55] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[56] Rongfei Zeng, Chao Zeng, Xingwei Wang, Bo Li, and Xiaowen Chu. A comprehensive survey of incentive mechanism for federated learning. *arXiv preprint arXiv:2106.15406*, 2021.

[57] Rongfei Zeng, Shixun Zhang, Jiaqi Wang, and Xiaowen Chu. Fmore: An incentive scheme of multi-dimensional auction for federated learning in mec. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 278–288. IEEE, 2020.

[58] Chaoning Zhang, Philipp Benz, Dawit Mureja Argaw, Seokju Lee, Junsik Kim, Francois Rameau, Jean-Charles Bazin, and In So Kweon. Resnet or densenet? introducing dense shortcuts to resnet. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pages 3550–3559, 2021.

[59] Zhaohua Zheng, Yize Zhou, Yilong Sun, Zhang Wang, Boyi Liu, and Keqiu Li. Applications of federated learning in smart cities: recent advances, taxonomy, and open challenges. *Connection Science*, pages 1–28, 2021.