CRACAU: Byzantine Machine Learning Meets Industrial Edge Computing in Industry 5.0

Anran Du, Yicheng Shen, Qinzi Zhang, Lewis Tseng, *Member, IEEE*, and Moayad Aloqaily, *Senior Member, IEEE*

***Abstract*—Industry 5.0 is emerging as a result of the advancement in networking and communication technologies, artificial intelligence, distributed computing, and beyond 5G. Among the important enabling technologies, federated learning, industrial edge computing, and Byzantine-tolerant machine learning (ML) are key accelerators in Industry 5.0. We propose a framework to integrate these key components. Recent works have designed various Byzantine-tolerant ML algorithms for a datacenter or a cluster. However, these algorithms are difficult to be applied to industrial edge computing paradigms. In this article, a novel Byzantine-tolerant federated learning algorithm, CRACAU, is designed for the popular three-level edge computing architecture. In this algorithm, edge devices jointly learn an ML model using the data collected at each device, and their private data are never shared with others. Under standard assumptions, we formally prove that CRACAU converges to the optimal point, i.e., CRACAU finds the optimal parameters of the ML model. We also implement CRACAU in the MXNet framework and evaluate it on the popular benchmark MNIST and CIFAR-10 image classification datasets. Experimental results show that CRACAU achieves satisfying accuracy.**

***Index Terms*—Artificial intelligence (AI), Byzantine fault tolerance, federated learning, industrial edge computing, Industry 5.0.**

I. Introduction

FUTURISTIC technologies have accelerated the emergence of Industry 5.0. Distributed computing, integration of artificial intelligence, all types of applications, privacy-preserving schemes, beyond 5G networks, and industrial Internet of things (IIoT) are the fuel for this drive. Such applications usually require huge bandwidth and computation demands along with stringent constraints on service latency [1], [2]. Hence, the edge computing paradigm is a promising architecture owing to three main advantages: 1) cloud server(s) in the datacenter is no longer the communication and computation bottleneck, because edges help handle these tasks; 2) since edge devices only communicate with local edge server(s), latency is greatly reduced; and 3) it improves efficiency by shifting the centralized system into more distributed one.

Another emerging trend in distributed industrial applications is the adoption and integration of machine learning (ML) algorithms. These applications may use ML algorithms for various purposes: 1) decision making as in smart environments and ambient intelligence; 2) pattern recognition as in object and scene understanding in augmented reality, and face recognition in real-time identity verification; and 3) prediction as behavior prediction in mobile gaming and autonomous vehicles, and viewport prediction in video streaming. Traditionally, these ML models are trained offline with historical and synthetic data. To handle dynamic workloads, recent works propose online ML training using real-time data.

ML meets industrial edge computing: For efficient online ML training, real-time data collection and computation tasks are distributed to both edge devices and/or edge servers, because it is too expensive and raises privacy concerns if the full datasets collected at devices are transmitted back to the cloud server(s). In the past few years, many frameworks have been proposed to integrate online ML training directly with edge or wireless devices, e.g., mobile phones, sensor networks, smart transportation, IIoT, etc. [3]. Several ML algorithms from both academia and industry are specifically designed for edge computing architectures, including edge learning framework [4], eSGD [5], fog learning [6], and federated learning on vehicular nodes [7].

We consider a standard three-level architecture for industrial edge computing, as depicted in Fig. 1. The bottom level consists of edge devices in Industry 5.0, e.g., Internet of things devices, sensor nodes, mobile phones, high-end vehicles, etc. The middle layer is the "edge or fog computing layer," which consists of the edge servers in the form of proxy servers, routers, road-side units, cloudlets, or base stations. These nodes are capable of providing essential storage, communication, and computation capability. The top level is the cloud computing layer, which consists of the cloud datacenter(s) and servers.

Fault-tolerant edge-based ML algorithms: In Industry 5.0, fault tolerance is the first-class citizen. This work lays out the theoretical foundation for fault-tolerant edge-based ML that tolerates Byzantine faults, the strongest type of fault model. We

Manuscript received April 1, 2021; revised June 15, 2021 and July 3, 2021; accepted July 9, 2021. Date of publication July 14, 2021; date of current version May 6, 2022. Paper no. TII-21-1498. *(Anran Du, Yicheng Shen, and Qinzi Zhang contributed equally to this work.) (Corresponding author: Moayad Aloqaily.)*

Anran Du, Yicheng Shen, Qinzi Zhang, and Lewis Tseng are with the Department of Computer Science, Boston College, Chestnut Hill, MA 02467 USA (e-mail: duab@bc.edu; shenvw@bc.edu; zhangbcu@bc.edu; lewis.tseng@bc.edu).

Moayad Aloqaily is with the Machine Learning Department, Mohamed Bin Zayed University of Artificial Intelligence (MBZUAI), Abu Dhabi 31859, UAE (e-mail: moayad.aloqaily@mbzuai.ac.ae).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TII.2021.3097072.

Digital Object Identifier 10.1109/TII.2021.3097072

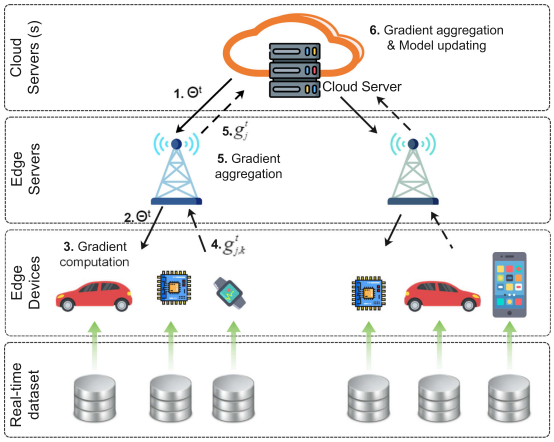


Fig. 1. Three-level architecture of edge computing. Section III-C presents the exact steps. The training proceeds in rounds, where θ^t and g^t are parameter and gradient (at edge devices or servers) in round t.

propose a novel Byzantine ML algorithm, *CRACAU* (collect–reduce–aggregate–collect–aggregate–update). It is designed for the three-level industrial edge computing architecture as in Fig. 1 and optimized for distributed systems in which edge devices collect real-time data at its current location and jointly aim to learn an online ML model. CRACAU is optimized for such a scenario in the sense that by utilizing the characteristics of location-based data, our algorithm is more efficient than prior Byzantine ML algorithms. CRACAU proceeds in rounds and achieves *computation and communication complexity* linear in the number of edge devices in each round, making it more scalable. Scalability is an important metric in edge-based MLs because there is a significant amount of edge devices, and edge servers are usually not as powerful as typical servers in datacenters. Our algorithm can be viewed as a form of federated learning [8], since each edge device does not share its own private data, and the training is only conducted on the devices.

Under standard assumptions [9], [10], we formally prove that CRACAU converges to the optimal point in the presence of Byzantine edge devices, i.e., CRACRU is guaranteed to find the optimal parameters of the ML model eventually if the cost functions and real-time datasets follow our assumptions. We also implement CRACAU in the MXNet,[1] a popular open-source ML framework. The MNIST and CIFAR-10 image classification datasets are used to evaluate CRACAU. Experimental results show that: 1) CRACAU has a satisfying accuracy even under Byzantine attack and 2) CRACAU converges with disjoint local dataset (non-independent identically distributed (non-i.i.d.) datasets). Finally, we discuss mechanisms to extend CRACAU to reduce communication complexity and enhance privacy and resilience.

Contributions: We have three contributions in this article.

1) We propose a novel Byzantine-tolerant edge-based FL algorithm—CRACAU.
2) We formally prove that CRACAU converges under standard assumptions on the cost function and datasets.
3) We implement CRACAU in MXNet and evaluate it using MNIST and CIFAR-10.

[1][Online]. Available: https://mxnet.apache.org/

TABLE I
CRACAU AND CLOSELY RELATED WORK

Name	Edge Computing	Byzantine-Tolerance	Formal Proof	Complexity
KRUM [10]		✓	✓	$O(md^2)$
ZENO [13]		✓	✓	$O(md)$
CGC [9]		✓	✓	$O(md + m \log m)$
Fog Learning [6]	✓			$O(md)$
eSGD [5]	✓			$O(md)$
FEDAVG [15]		✓	✓	$O(md)$
CRACAU	✓	✓	✓	$O(md)$

The rest of this article is organized as follows. Section II presents related work. Preliminaries and assumptions are in Section III. Our algorithm is proposed in Section IV, and proof shown in Section V. We discuss the evaluation in Section VI. Finally, Section VII concludes this article.

II. STATE OF THE ART

ML has rapidly gained popularity in both academia and industry recently. New paradigms of distributed ML architectures have been proposed to handle the exponential growth in the training data size and algorithm and model complexity [3], [11]. Due to space constraint, we only discuss the most relevant works here. Our work is inspired by the recent advance in Byzantine ML and edge-based ML algorithms [12]. CRACAU is the first edge-based ML designed to tolerate Byzantine faults.

Table I compares closely related ML algorithms with CRACAU. In the table, we first identify if the corresponding algorithm is designed for datacenters (and clusters) or edge computing. Then, we point out if the algorithm tolerates Byzantine faults. Third, we mark algorithms that are proven to converge mathematically. Finally, we identify the per-round computation complexity, where m is the number of servers or edge devices, and d is the number of dimensions in the ML model. All the algorithms in the table proceed in synchronous rounds. Zeno [13] and Adaptive FEDAVG [14] are efficient because of the assumption of failure detector. We do *not* adopt such model because the efficacy of failure detectors is not clear in edge computing. CRACAU is able to achieve the same per-round efficiency with our novel design.

A. Byzantine-Tolerant ML

In the fault-tolerant and distributed computing community, malicious parties are usually modeled by "Byzantine faults," where faulty nodes are controlled by a computationally unbounded adversary that can behave arbitrarily. Byzantine ML algorithms converge to the optimal point of the ML model even if some workers become Byzantine. All the Byzantine ML algorithms that we are aware of are designed for datacenters [9], [10],

[13]. Krum [10] is one of the first Byzantine-tolerant stochastic gradient descent (SGD) algorithms. Gupta and Vaidya [9] studied the general distributed optimization framework. Xie *et al.* [13] proposed Zeno, which uses a failure detector to tolerate more faults. Adaptive FEDAVG [15] is a fault-tolerant federated learning algorithm that also relies on some form of a failure detector. None of these works were designed for edge computing architectures. Moreover, as identified in Table I, our algorithm CRACAU has the best per-round computation complexity.

B. Edge-Based ML

In typical distributed applications deployed in the edge computing paradigms, the edge devices are usually small and equipped with only limited battery capacity. One important metric is the communication complexity, since expensive and frequent communication drain device battery quickly. Several works have focused on reducing per-round communication complexity and limiting the communication to mostly edge-device communications (instead of the wide-area communication between devices and cloud servers). These algorithms are *not* designed to tolerate Byzantine faulty workers, and they use entirely different approaches to reduce communication. For example, eSGD [5] discards coordinates of the local gradients at each edge server aggressively, and fog learning [6] uses intermediate aggregation. No formal analysis on convergence was presented in [5] and [6]; therefore, there is no clear way to integrate Byzantine masking mechanisms with these algorithms.

III. PRELIMINARIES

We formally define the models and framework and present preliminaries on ML and Byzantine faults.

A. Stochastic Gradient Descent

We explore the Byzantine-tolerant SGD algorithms [9], [10]. We formally define the SGD framework below. The SGD algorithm is designed to produce an optimal parameter θ^* in a d-dimensional space that minimizes the cost function given a cost function Q, i.e.,

$$\theta^* = \operatorname*{argmin}_{\theta \in \mathbb{R}^d} Q(\theta). \tag{1}$$

An SGD algorithm is designed to execute iteratively. That is, in each round t, two tasks need to be completed: (i) computing the gradient of the cost function Q at parameter θ^t and (ii) using gradient descent to update the parameter (of the ML model).

In our distributed setup, each edge server j has a local cost function Q_j, and the distributed SGD algorithm aims to minimize the overall cost function and compute θ^* such that the sum of all the cost functions is minimized. That is, we aim to solve the following equation:

$$\theta^* = \operatorname*{argmin}_{\theta \in \mathbb{R}^d} \sum_{j=1}^{n} Q_j(\theta). \tag{2}$$

B. Synchronous Parameter Server Model in Edge Computing

It is expensive to compute gradients on the entire dataset. One popular solution is using the *parameter server model* framework to parallelize the computation. In each round, the parameter server assigns (gradient) calculation tasks to all jobs [task (i)] and then aggregates their computed gradients to update the ML model's parameter [task (ii)].

In our three-level edge computing architecture (as in Fig. 1), both edge servers and the cloud server need to perform the aggregation, but only the cloud server will update the parameter. In the presentation below, we use "workers" and "edge devices" interchangeably, since they are the ones performing gradient computation tasks.

This article considers a *synchronous* system. That is, each worker and server proceed in a lock step. Later, in Section IV-C, we discuss how to relax the assumption on synchrony.

C. SGD in Edge Computing

A distributed SGD algorithm in our three-level architecture proceeds in rounds synchronously and follows the following steps for each round $t \geq 0$.

1) The cloud server sends parameter θ^t to all edge servers.
2) The edge server j forwards θ^t to nearby workers. Since workers are mobile nodes, we denote each such worker as $w_{j,k}^t$, i.e., the kth worker that does the gradient computation for edge server j in round t.
3) Each worker $w_{j,k}^t$ randomly chooses a data batch $\xi_{j,k}^t$ from its local dataset (that are collected using on-device sensors) and computes a local estimate gradient, $g_{j,k}^t$. Following the convention (see, e.g., [10]), the computation is quantified by a local estimator function F, i.e., $g_{j,k}^t$ is obtained by computing $F(\theta^t, \xi_{j,k}^t)$.
4) Worker $w_{j,k}^t$ then sends $g_{j,k}^t$ back to the edge server j. Byzantine faulty worker may send an arbitrary gradient.
5) Upon collecting enough gradients from nearby workers, the edge server j computes and sends an aggregated gradient g_j^t to the cloud server.
6) Upon collecting aggregated gradients from all the edge servers, the cloud server then updates the parameter with step size η as follows:

$$\theta^{t+1} = \theta^t - \eta \sum_{j=1}^{n} g_j^t. \tag{3}$$

A similar edge-ML framework is adopted in [5] and [6].

D. Fault Model and Byzantine SGD

For brevity, we assume that the central cloud server and edge servers are always *fault-free*, i.e., they do not crash and follow the algorithm specification faithfully. Later, in Section VII, we discuss how to relax this assumption. Since edge devices are more fragile, we adopt the local fault model [16]. That is, up to a fraction of workers close to an edge server may become Byzantine faulty.

TABLE II
MOST USED NOTATIONS

t	round index
θ^t	parameter at round t, for $t \geq 0$
n	number of edge servers
j	index of edge server j, $j = 1, 2, \ldots, n$
m_j	number of required gradients collected at edge server j
$w_{j,k}^t$	k-th worker at edge server j, for each $k = 1, 2, \ldots, m_j$
f_j	maximum number of Byzantine workers at edge server j
C_j^t	set of workers communicating with edge server j in round t
$\mathcal{H}_j^t$	set of fault-free workers in C_j^t
$\mathcal{B}_j^t$	set of Byzantine workers in C_j^t
h_j^t	$h_j^t = \lvert\mathcal{H}_j^t\rvert$; $\quad m_j - f_j \leq h_j^t \leq m_j$
b_j^t	$b_j = \lvert\mathcal{B}_j^t\rvert$; $\quad b_j^t \leq f_j$
$r_{\max}$	upper bound of f_j/m_j, for all j
Q_j	cost function of edge server j
L, μ	Lipschitz constant and strong convexity constant
$\xi_{j,k}^t$	a data batch used by worker $w_{j,k}^t$ in round t

Formally, we assume that there are n edge servers and denote the index of an edge server as j, for $j = 1, 2, \ldots, n$. In step 5), each j will collect m_j gradients from nearby workers, where m_j is a configurable parameter. Among these m_j collected gradients, at most f_j are from Byzantine workers. The fraction of Byzantine workers, $\frac{f_j}{m_j}$, is assumed to be bounded by a constant $r_{\max}$ for the edge server j. The bound on $r_{\max}$ is analyzed in Section V.

Byzantine workers may have arbitrary behaviors, including manipulating its local dataset and estimator function F. Faulty workers may work with each other and have a complete knowledge (an omniscient adversary model). We aim to design a distributed SGD algorithm that solves (2) when $\leq r_{\max}$ fraction of Byzantine workers at each edge server.

Workers that follow the protocol specification are called fault-free nodes. For each edge server j at round t, we denote C_j^t whose gradients are collected and used by j in round t. By definition, $|C_j^t| = m_j$. We denote $\mathcal{H}_j^t$ as the set of fault-free workers in C_j^t and $\mathcal{B}_j^t$ as the set of Byzantine workers in C_j^t; we also denote $h_j^t = |\mathcal{H}_j^t|$ and $b_j^t = |\mathcal{B}_j^t|$. By definition, we have $b_j^t \leq f_j$ and $h_j^t \geq m_j - f_j$. Most important notations used in the discussion are presented in Table II.

IV. PROPOSED ALGORITHM: CRACAU

In Section III-C, we present the framework of edge-based ML algorithm that uses SGD. One key challenge to tolerate Byzantine faults is at step 5) when edge servers need to mitigate faulty gradients sent by the Byzantine workers. We devise a filter called *Reduce filter*, which is inspired by a popular mechanism from a series of Byzantine consensus works, e.g., [17], and the CGC filter in Byzantine ML [9]. Because of the different assumptions on the cost functions and datasets used (which we will formally define later in Section V), our Reduce filter is simpler and has a smaller computation complexity than the CGC filter [9].

The name of our algorithm, CRACAU, summarizes the key computation steps at the edge servers (collect–reduce–aggregate) and the cloud server (collect–aggregate–update).

Algorithm 1: Algorithm CRACAU.

Parameters:
  n is the number of edge servers
  m_j is the number of required gradients collected at each edge server j
  $\eta > 0$ is the step size defined in (3)
Initialization at cloud server:
  $\theta^0 \leftarrow$ a random vector in $\mathbb{R}^d$ ▷ initial parameter
Main Algorithm:
for round $t \leftarrow 0$ to ∞ **do**
  ===== **Steps at Cloud Server** =====
    send θ^t to all edge servers $\{1, 2, \ldots, n\}$
  ===== **Steps at Worker** $w_{j,k}^t$ =====
    retrieve θ^t from the nearby edge server j
    ▷ Compute local gradient using an estimator function F on data batch $\xi_{j,k}^t$
    $g_{j,k}^t \leftarrow F(\theta^t, \xi_{j,k}^t)$
    send $g_{j,k}^t$ to edge server j
  ===== **Steps at Edge Server** j =====
    ▷ *Collect Step*
    Wait until j has collected local gradients from m_j distinct workers
    $G_j \leftarrow$ the set of collected gradients
    ▷ *Reduce Step*
    $G_j' \leftarrow Reduce(G_j)$
    ▷ *Aggregate Step*
    $g_j^t \leftarrow \sum_{g' \in G_j'} g'$
    send g_j^t to the cloud server
  ===== **Steps at Cloud Server** =====
    ▷ *Collect Step*
    Wait until cloud server has collected aggregated gradients from all n edge servers
    ▷ *Aggregate Step*
    $g^t \leftarrow \sum_{j=1}^n g_j^t$
    ▷ *Update Step*
    $\theta^{t+1} \leftarrow \theta^t - \eta g^t$
end for

Recall that we assume a *synchronous* parameter server framework for the three-level edge computing architecture, as depicted in Section III-B. Hence, CRACAU is presented as an iterative algorithm. That is, workers, edge servers, and the cloud sever proceed in rounds synchronously. Algorithm 1 presents the pseudocode of CRACAU. Table II summarizes important notations and constants used in the algorithm.

A. CRACAU: Description

Initially, the cloud server generates an initial parameter $\theta^0 \in \mathbb{R}^d$ for the specific ML model randomly. In each round $t \geq 0$, the cloud server first disseminates the current parameter θ^t to all the n edge servers. Each edge server $j \in \{1, 2, \ldots, n\}$ then forwards θ^t to nearby workers. Each j is configured with a fixed

parameter, m_j, which denotes the number of gradients that need to be collected at j in each round t. The parameter forwarding from j to the workers can also be achieved using a hybrid of push and pull mechanisms [5]. Up to this point, the steps are similar to non-fault-tolerant edge-based ML in [5] and [6]. We next describe our key innovation.

1) Fault Tolerance at Edge Servers: Upon receiving the parameter, each worker $w_{j,k}^t$ computes the local gradient using an estimator function F on the data batch $\xi_{j,k}^t$, which is a set of sampled data points from the dataset collected by $w_{j,k}^t$ at location near the edge server j in round t. Then, the worker sends the computed estimated gradient $g_{j,k}^t$ to the edge server j. The reason that we call it an "estimated gradient" is that the gradient is an estimation of the true gradient of the cost function at the edge server j, Q_j. Later, in Section V, we will formally define the relation between Q_j and $g_{j,k}^t$ so that CRACAU is able to converge.

We now describe the CRACAU steps. The first part is for each edge server j.

1) *Collect step:* j collects m_j local estimated gradients from distinct workers. These gradients need to come from distinct devices so that Byzantine workers can transmit at most one faulty gradient.
2) *Reduce step:* Denote by G_j the set of gradients that j collects in the previous step. Here, we drop the round index t for brevity. Edge server j then uses the Reduce filter on G_j to mitigate the impacts from Byzantine workers. Concretely, the steps to compute Line 1 in Algorithm 1, $G'_j \leftarrow Reduce(G_j)$, are as follows.
   a) Compute the Euclidean norm of the gradients in G_j.
   b) Find the gradient x that has the f_jth largest Euclidean norm in G_j. Break tie using the lexicographical ordering.
   c) Discard f_j gradients with the largest Euclidean norm. Formally, j examines each gradient g in G_j. If $\|g\| \geq \|x\|$, then j discards the gradient g; otherwise, j adds g into the set G'_j. Here, $\|\cdot\|$ denotes the Euclidean norm of a gradient.
   d) Output G'_j.
3) *Aggregate step:* j then aggregates the remaining gradients that are *not* discarded by the Reduce filter. Formally, j's aggregated gradient in round t is computed as
$$g_j^t \leftarrow \Sigma_{g' \in G'_j} g'.$$

2) Parameter Server at Cloud Server: The cloud server also has three steps, which is similar to how typical parameter servers operate, e.g., [5], [6], [10].

1) *Collect step:* The cloud server collects aggregated gradients from all the n edge servers.
2) *Aggregate step:* The cloud server aggregates the gradients from edge servers by summing them together. Denote the derived gradient as g^t.
3) *Update step:* The cloud server updates the parameter using the SGD method on the current parameters θ^t, g^t, and the step size η as $\theta^{t+1} \leftarrow \theta^t - \eta g^t$.

The aggregate and update steps capture the SGD method specified in (3).

B. Properties

We prove that CRACAU converges to the optimal parameter under the standard assumptions of Byzantine ML (see, e.g., [9] and [10]) in Section V. We summarize key properties as follows.

1) Feasibility: With a properly configured size of data batch and ML model, the computation in line 1 (computation of the estimator function F) can, in fact, be completed in the matter of seconds to minutes on a normal laptop using a single core in our experience. Modern edge devices such as high-end vehicles, mobile phones, and even smart watches are gradually equipped with specialized ML chips, which should outperform one core in a laptop in ML training. Hence, we believe that it is appropriate to assume that each worker $w_{j,k}^t$ is able to complete the training when it is still in the communication range with the edge server j. In Section IV-C, we discuss how to configure data batch and sketch schemes to allow workers to upload its own estimated gradient to another edge server.

2) Linear Computation: CRACAU has time complexity linear with respect to the number of works in each round t. More precisely, the computation complexity in each round is linear with respect to the number of workers. The collect steps, aggregate steps, and update step all take linear time. The only remaining step is the Reduce filter. It takes linear time to compute Euclidean norm discard gradients. Moreover, we use the QUICKSELECT algorithm [18] to find the gradient x in the unsorted set G_j. QUICKSELECT is able to find the kth smallest (or largest) element in an unsorted set and has a linear time complexity with respect to the number of elements in the set. Since in CRACAU, each worker submits one gradient, the size of all the G_j's are exactly the number of workers. Therefore, the time complexity of CRACAU is linear.

3) Communication Complexity: In each round, each worker and each edge server only need to transmit one gradient. Hence, CRACAU has a linear communication complexity. Since each edge server is assumed to be fault-free, we can further reduce more communication using the approaches similar to [5]. Roughly speaking, after the Reduce filter, each aggregated gradient computed by the edge server is guaranteed to provide enough information to perform SGD with high probability. Therefore, we can suppress the transmission of some gradients from edge servers or some dimensions in the gradients.

4) Privacy: CRACAU can be viewed as a form of federated learning [8], since all the edge devices keep their private data. That is, the ML training is conducted locally on each worker, and no private data are ever sent to the third party, i.e., edge servers and cloud server. Later, in Section VII, we briefly discuss how to extend CRACAU so that we can provide a certain level of location privacy.

C. Discussion

We present practical improvements and extensions.

1) *Configuring m_j:* If the number of surrounding workers is less than m_j, j needs to wait for more workers to move closer and complete the gradient computation. One can properly configure m_j based on the traffic pattern (the number of surrounding workers) so that the impact to

latency due to lack of workers is minimal. There is an inherent tradeoff between traffic patterns, fault tolerance, and convergence speed.

2) *Identifying fundamental limits of non-i.i.d. limitation:* It is generally believed to be impossible to design a distributed Byzantine ML algorithm in a truly non-i.i.d. assumption (as implied by the impossibility result in [9]), because a Byzantine adversary may have an arbitrary behavior; hence, it is necessary to have some redundancy (as in a typical i.i.d. model or our local i.i.d. model) to mask faulty tampering. An interesting direction is to use some external knowledge, e.g., more assumption on the data or some form of failure detectors [13], to design an efficient ML algorithm.
3) *Incorporating asynchronous update rules with reduced communication complexity and improved speed:* The current specification (line 1 in Algorithm 1) requires that the cloud server waits and collects the gradients from *all* the edge servers. However, this can be relaxed by collecting aggregated gradients from *only a proportion* of edge servers. This will affect the convergence speed because, intuitively, we lose potentially important "information" by collecting a smaller set of gradients. However, this does not affect the overall convergence. This extension allows us to make CRACAU asynchronous. Moreover, edge servers without a sufficient number of workers can also be neglected in this round.

   One limitation of CRACAU is that each worker $w_{j,k}^t$ can only send its gradient to the edge server j. If the worker moves out of the communication range, then its computation is wasted. There are two mechanisms to fix the issue. First, the worker can send to any edge server j', whose cost function is similar, i.e., $Q_j \approx Q_{j'}$. This approach allows us to apply essentially the same argument for convergence and requires a minimal change to the algorithm. This approach also provides a notion of location privacy because each worker does not need to reveal its entire location history, whereas, in CRACAU, workers need to communicate with and reveal location to the edge servers in each round. This extension provides a form of a-anonymity, where a denotes the number of edge servers sharing similar cost functions.

   The first approach, however, requires workers to know the cost functions, which is not always available. Hence, we propose the second approach, in which edge servers need to forward gradients to the original edge server j so that j can apply the aggregation with enough information. One slight drawback is that this approach does not tolerate the local fault model [16], [19] because Byzantine workers can target a specific edge server j and "flood" all the faulty gradients to j. Some form of failure detector is necessary to mitigate this attack, i.e., discarding gradients that are too far off.
4) *Addressing the catastrophic forgetting issue:* If the pattern changes drastically, then the learned parameters (of the ML model) need to be forgotten and relearned. This can potentially be achieved by the standard method, such as integrating a quadratic penalty term in the loss function. Granted, impact of Byzantine faults is an unexplored area in the literature, and more studies need to be done empirically to verify the efficacy.

V. Convergence Proof

We assume a synchronous system. Hence, our analysis is focused within a single round. We then apply the results inductively to each round $t \geq 1$. For brevity, in the discussion below, we will omit the subscript t.

A. Assumptions

Recall that Q_j is the cost function at each edge server j. We assume that Q_j satisfy standard properties [10], [20], [21], such as strong convexity, Lipschitz smoothness, and differentiability, as defined in the following. We also adopt standard notations: $\langle a, b\rangle$ has been used to represent the inner product of the two vectors a and b in the d-dimensional space $\mathbb{R}^d$.

Assumption 1 (Convexity and smoothness): The variable Q_j is assumed to be convex and differentiable.

Assumption 2 (L-Lipschitz smoothness): For all $\theta, \theta' \in \mathbb{R}^d$ and for all j, there exists $L > 0$

$$\|\nabla Q_j(\theta) - \nabla Q_j(\theta')\| \leq L\|\theta - \theta'\|.$$

Assumption 3 (μ-strong convexity): There exists $\mu > 0$ such that for all $\theta, \theta' \in \mathbb{R}^d$ and for all j,

$$\langle \nabla Q_j(\theta) - \nabla Q_j(\theta'), \theta - \theta'\rangle \geq \mu\|\theta - \theta'\|^2.$$

We also render the i.i.d. statement, which states that data batches are distributed from the dataset separately and identically at each worker. Formally speaking, we denote an operator $\mathbb{E}_{\Xi^t|\theta^t}$ as the *conditional expectation* operator over the set of random batches of all workers in round t, i.e., $\Xi^t = \{\xi_{j,k}^t\}$, given the parameter θ^t. We analyze the conditional expectation given θ^t, because it allows us to treat θ^t, as well as $Q_j(\theta^t)$ and $\nabla Q_j(\theta^t)$, as known constants. For brevity, we abbreviate the operator $\mathbb{E}_{\Xi^t|\theta^t}$ as $\mathbb{E}$ in the presentation below.

With the definition of $\mathbb{E}$, we present two assumptions on the properties of the data batch $\xi_{j,k}^t$ and the estimator function F.

Assumption 4 (i.i.d. random batches): For all j, k, and t, $\mathbb{E}(g_{j,k}^t) = \nabla Q_j(\theta_j^t)$.

Assumption 5 (Bounded variance): For all j, k, and t, $\mathbb{E}\|g_{j,k}^t - \nabla Q_j(\theta^t)\|^2 \leq \sigma^2\|\nabla Q_j(\theta^t)\|^2$ and $\sigma^2 < \min_j(\frac{1}{m_j})$.

Following [9], we assume that the local cost functions at edge servers also satisfy the $2f$-redundant property. More specifically, we assume the following is true.

Assumption 6: There exists an optimal parameter $\theta^* \in \mathbb{R}^d$ such that $\theta^* = \text{argmin} Q_j(\theta)$ for all j.

Assumption 6 implies that such θ^* is also the optimal parameter $\theta^* = \text{argmin} \sum_{j=1}^n Q_j(\theta)$.

B. Convergence Analysis

Define θ^* as in Assumption 6. By gradient descent, we have

$$\begin{aligned}\mathbb{E}\|\theta^{t+1}-\theta^*\|^2 &\le \mathbb{E}\|\theta^t-\theta^*-\eta g^t\|^2\\ &= \mathbb{E}\|\theta^t-\theta^*\|^2 \qquad \text{(Part A)}\\ &\quad -2\eta\mathbb{E}\langle\theta^t-\theta^*,g^t\rangle \qquad \text{(Part B)}\\ &\quad +\eta^2\mathbb{E}\|g^t\|^2. \qquad \text{(Part C)}\end{aligned} \tag{4}$$

We will analyze each part separately.

Part A: By definition of $\mathbb{E}$, we can treat θ^t as a known constant, so $\mathbb{E}\|\theta^t-\theta^*\|^2=\|\theta^t-\theta^*\|^2$.

Part B: Define G' as in line (24) of Algorithm 1, i.e., the set of reduced gradients. By linearity of expectation, we have

$$\begin{aligned}\mathbb{E}\langle\theta^t-\theta^*,g^t\rangle &= \sum_{j=1}^{n}\mathbb{E}\langle\theta^t-\theta^*,g_j\rangle\\ &= \sum_{j=1}^{n}\sum_{g\in G'_j}\mathbb{E}\langle\theta^t-\theta^*,g\rangle.\end{aligned} \tag{5}$$

Our proofs rely on the following lemmas proved in prior work [22]. We say that a gradient is faulty if it is originated from a Byzantine worker. Similarly, if a gradient is from a fault-free worker, then we say that the gradient is fault-free.

Lemma 1: For $x\ge 1$, define $k_x=1+\frac{x-1}{\sqrt{2x-1}}$. Then, there exists k^* such that $k_x/\sqrt{x}\le k^*, \forall x\ge 1$, and $k^*\approx 1.12$.

Lemma 2: For all j,t, if $g\in G'_j$ is fault-free, then

$$\mathbb{E}\|g\|\le(1+\sigma)L\|\theta^t-\theta^*\|.$$

Otherwise, if $g\in G'_j$ is faulty, then

$$\mathbb{E}\|g\|\le(1+k_{h_j}\sigma)L\|\theta^t-\theta^*\|.$$

By Lemma 2, for each faulty $g\in G'_j$,

$$\begin{aligned}\mathbb{E}\langle\theta^t-\theta^*,g\rangle &\ge -\|\theta^t-\theta^*\|\|g\|\\ &\ge -(1+k_{h_j}\sigma)L\|\theta^t-\theta^*\|^2.\end{aligned} \tag{6}$$

By Assumption 4, for any fault-free $g\in G'_j$, $\mathbb{E}(g)=\nabla Q_j(\theta^t)$. By strong convexity (Assumption 3),

$$\begin{aligned}\mathbb{E}\langle\theta^t-\theta^*,g\rangle &= \langle\theta^t-\theta^*,\mathbb{E}(g)\rangle\\ &= \langle\theta^t-\theta^*,\nabla Q_j(\theta^t)\rangle\\ &\ge \mu\|\theta^t-\theta^*\|^2.\end{aligned} \tag{7}$$

Note that: 1) $|G'_j|=m_j-f_j$; 2) $h'_j\ge h_j-f_j$; and 3) $b'_j\le b_j$, where we denote h'_j as the number of faulty-free gradients in G'_j and $b'_j=|G'_j|-h'_j$. Upon combining (6) and (7) into (5), we obtain that

$$\begin{aligned}\mathbb{E}\langle\theta^t-\theta^*,g\rangle &\ge \sum_{j=1}^{n}\begin{pmatrix}h'_j\left(\mu\|\theta^t-\theta^*\|^2\right)+\\ b'_j\left(-(1+k_{h_j}\sigma)L\|\theta^t-\theta^*\|^2\right)\end{pmatrix}\\ &\ge \sum_{j=1}^{n}\mathcal{K}_j\|\theta^t-\theta^*\|^2\end{aligned} \tag{8}$$

where $\mathcal{K}_j=(h_j-f_j)\mu-b_j(1+k_{h_j}\sigma)L$.

Part C: By convexity of $\|\cdot\|^2$, we have

$$\|g\|^2\le n\sum_{j=1}^{n}\|g_j\|^2,\quad \|g_j\|^2\le|G'_j|\sum_{g\in G'_j}\|g\|^2. \tag{9}$$

Denote $\alpha_x=x\sigma^2+(1+k_x\sigma)^2$. We introduce another lemma from [22], which states that $\mathbb{E}\|g\|^2$ is also bounded.

Lemma 3: For all j,t, if $g\in G'_j$ is fault-free, then

$$\mathbb{E}\|g\|^2\le\alpha_1L^2\|\theta^t-\theta^*\|^2.$$

Otherwise if $g\in G'_j$ is faulty, then

$$\mathbb{E}\|g\|^2\le\alpha_{h_j}L^2\|\theta^t-\theta^*\|^2.$$

Thus, (9) gives

$$\mathbb{E}\|g^t\|^2\le n\sum_{j=1}^{n}m_j(h_j\alpha_1+b_j\alpha_{h_j})L^2\|\theta^t-\theta^*\|^2. \tag{10}$$

Conclusion: Upon putting all three parts together, we obtain the final result that

$$\mathbb{E}\|\theta^{t+1}-\theta^*\|^2\le\rho\|\theta^t-\theta^*\|^2 \tag{11}$$

where $\rho=1-2\beta\eta+\gamma\eta^2$, and

$$\beta=\sum_{j=1}^{n}(h_j-f_j)\mu-b_j(1+k_{h_j}\sigma)L \tag{12}$$

and

$$\gamma=nL^2\sum_{j=1}^{n}m_j(h_j\alpha_1+b_j\alpha_{h_j}). \tag{13}$$

C. Convergence of CRACAU

Assumptions on fault tolerance $r_{\max}$: With some arithmetic operations, we can obtain a sufficient condition as follows:

$$\mu m_j-4.12Lf_j>0. \tag{14}$$

Finally, recall that for all j, $f_j/m_j<r_{\max}$. Hence, we can conclude that the following condition is a sufficient condition for $\beta>0$:

$$r_{\max}<\frac{1}{4.12(L/\mu)}. \tag{15}$$

Step-size η and convergence rate: Given $\beta,\gamma>0$, the quadratic function $\rho(\eta)=\gamma\eta^2-2\beta\eta+1$ reaches minimum at $\eta=\beta/\gamma$. Moreover, observe that $\rho(0)=1$. These two observations together with the property of a quadratic function imply that for all $\eta\in(0,2\beta/\gamma)$, $\rho\in[\rho(\beta/\gamma),1)$.

We can analyze the fastest convergence rate

$$\rho(\beta/\gamma)=1-\beta^2/\gamma. \tag{16}$$

Again, by some operations, we have

$$\rho\in[0,1),\quad \text{for}\quad 0<\eta<2\beta/\gamma. \tag{17}$$

This proves the convergence due to (11) and (17).

VI. Performance Evaluation

This section presents our evaluation using our customized simulator. For evaluation, we develop a customized Python framework that implements CRACAU using MXNet (one of the most popular ML frameworks) and integrates SUMO[2] to simulate the mobility pattern of the edge devices. Our implementation including the convolutional neural network (CNN) architecture will be released once the article is accepted. Two key findings are: 1) CRACAU tolerates various attacks and 2) CRACAU achieves a satisfying accuracy when training with disjoint local data, i.e., data batches at each worker follow a non-i.i.d. distribution. Our code can be found at our Github repository[3].

A. Evaluation

Our experiments are based on two popular benchmark datasets, the MNIST handwritten digits [23], which contains 60 k training examples, and a test set of 10 k examples, and the CIFAR-10 image classification dataset [24], which consists of 50 k training images and 10 k testing images. For the MNIST, our ML model is a sequential model with three dense layers. For the CIFAR-10, our ML model is a CNN with four convolutional layers followed by one fully connected layer. Experiments are conducted on virtual machines (VMs) of four cores and 16-GB memory on the Google cloud platform. We especially choose a small VM to model the computation limitation in not-so-powerful devices in Industry 5.0.

To simulate real-time dataset at each edge device, we first randomly form data batches of size 100. Then, we implement a queue at each edge server j that contains a fixed amount of the data batches. In the algorithm execution, whenever an edge device passes by j, the device will fetch a data batch from j's queue. We use both i.i.d. distribution and *disjoint local* datasets (or non-i.i.d. datasets). In i.i.d. distribution, data batches contain data points that are sampled in an i.i.d. fashion from the original dataset. We also evaluate a more realistic case with *disjoint local* datasets, where data batches at each queue are drawn from a different distribution. Concretely, a queue is configured to contain a majority of a certain type of data batches that consist of a specific category of digits or images and randomly draw digits or images from other categories uniformly at random. For example, data batches at edge server i contain a majority of automobile images, and data batches at edge server j contain a majority of truck images. Most prior Byzantine ML algorithms are designed for i.i.d. datasets, e.g., [10].

Each run of our experiments consists of 200 epochs. Following the convention (see, e.g., [5] and [13]), one epoch is defined as the duration when the entire dataset has been trained exactly once. Unless specifically mentioning a different setting, we launch eight edge servers with $m_j = 10$. The learning rate is set to 5×10^{-3} and 5×10^{-4} for the MNIST dataset and the CIFAR-10 dataset, respectively. For each experiment, we repeat four runs with a different random seed each time and report the average.[4] As is standard, top-1 accuracy on the testing set and the cross-entropy loss function on the training set are used as the evaluation metrics. The accuracy is evaluated by comparing the prediction made by our trained ML model with the test set.

Baseline: Mean . Since we are not aware of any Byzantine-tolerant edge-based ML, we use Mean as the baseline, which is a more robust version of Fog Learning [6]. In Mean, the edge server does not discard any gradient collected from workers. Instead, it takes an average as the aggregated gradient, i.e., the Reduce and Aggregate steps are replaced by a step that takes the mean of the workers' gradient.

Attacks. It turned out that Mean is quite robust, which survives popular attacks, such as Gaussian noise, label-flipping and bit-flipping, used in prior works such as [13] and [15]. Hence, we devise a more severe attack: *sign-flipping attack*. In the attack, we flip the sign of chosen layer(s) of the CNN and then multiple the magnitude by a factor.

Due to space constraints, we only report the results under the sign-flipping attack. CRACAU converges in the presence of all the attacks reported in [13].

B. Evaluation Result

Following [5], we first evaluate CRACAU using the MNIST dataset [23] with i.i.d. data batches. Fig. 2 presents the results under sign-flipping attack. As expected, CRACAU converges and achieves final accuracy 97.55%. Fig. 3 reports the results for the CIFAR-10 dataset with i.i.d. data batches. CRACAU achieves 75.49% accuracy even under sign-flipping attack. Mean achieves a much lower accuracy under attack. When the adversary "flips" more layers, Mean achieves roughly 10% accuracy.

For the non-i.i.d. data batches (disjoint local data), CRACAU also achieves a satisfying accuracy, i.e., it has comparable performance in the presence of sign-flipping attack to the baseline case (Mean's performance when there is no failure). For the MNIST, we have ten edge servers, whereas for the CIFAR-10, we have five edge servers. The results are plotted in Figs. 4 and 5, respectively. We do not report Mean's performance under attack, since it could not tolerate the attack with i.i.d. data batches, and the non-i.i.d. case is even harder to learn. CRACAU achieves 96.83% and 72.86% accuracy for MNIST and CIFAR-10, respectively.

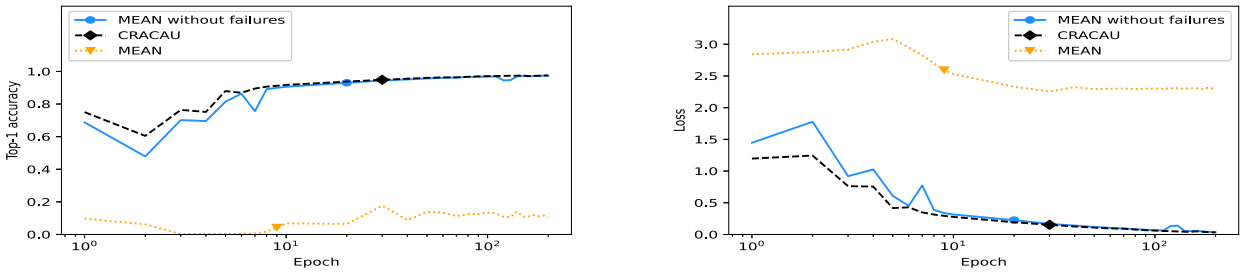
C. Krum [10] Versus CRACAU

We compare the accuracy and running time against a popular competitor, Krum [10]. Particularly, we apply the Krum filter at the edge server, with the rest being the same in our framework. Fig. 6 shows that both the edge version of Krum and CRACAU converge using the MNIST dataset in both the i.i.d. and non-i.i.d settings. Fig. 7 presents the accumulated time (over number

[2]Simulation of Urban MObility. [Online]. Available: www.eclipse.org/sumo/

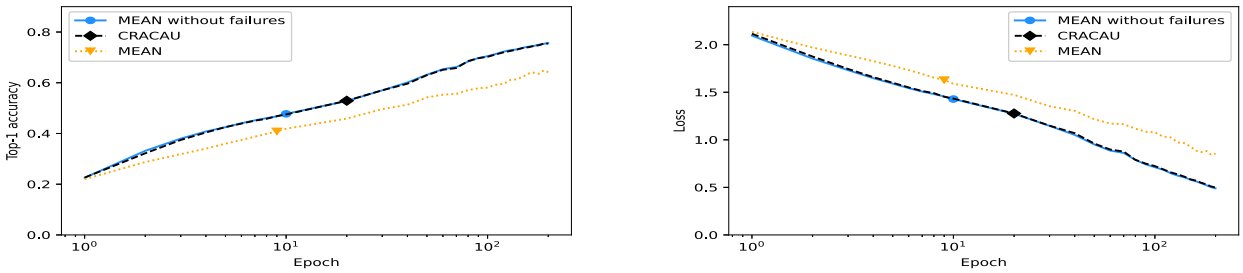
[3][Online]. Available: https://github.com/YichengShen/CRACAU

[4]Throughout our experiments, we have observed that the performance is quite stable even under various attacks. Hence, we believe that repeating four times is more than enough, since training usually takes a significant amount of time. This is also a convention in ML literature. For example, the experiments were only conducted three times and only the max was reported in [5].



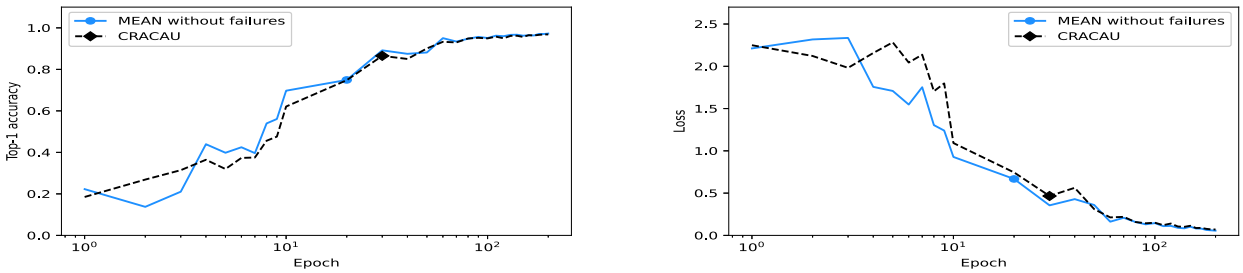
(a) (b)

Fig. 2. Convergence of CRACAU on the MNIST dataset (i.i.d. data batches and sign-flipping attack). (a) Top-1 accuracy on the MNIST testing set. (b) Cross entropy on the MNIST training set.



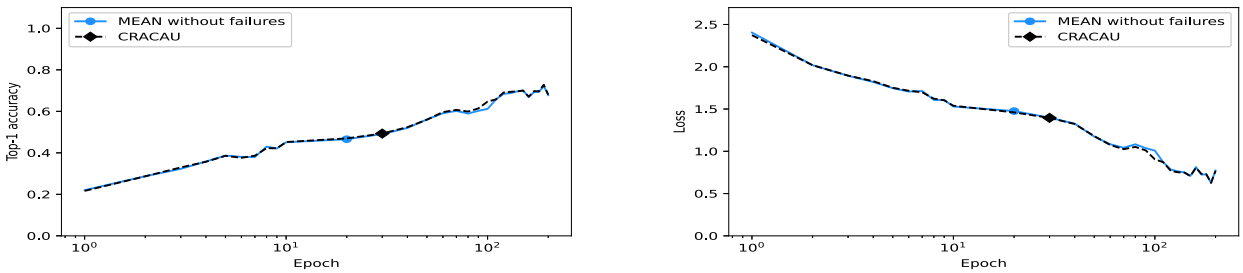
(a) (b)

Fig. 3. Convergence of CRACAU on the CIFAR-10 dataset (i.i.d. data batches and sign-flipping attack). (a) Top-1 accuracy on the CIFAR-10 testing set. (b) Cross entropy on the CIFAR-10 training set.



(a) (b)

Fig. 4. Convergence of CRACAU on disjoint local data from the MNIST dataset (non-i.i.d. data batches and sign-flipping attack). (a) Top-1 accuracy on the MNIST testing set. (b) Cross entropy on the MNIST training set.



(a) (b)

Fig. 5. Convergence of CRACAU on disjoint local data from the CIFAR-10 dataset (non-i.i.d. data batches and sign-flipping attack). (a) Top-1 accuracy on the CIFAR-10 testing set. (b) Cross entropy on the CIFAR-10 training set.

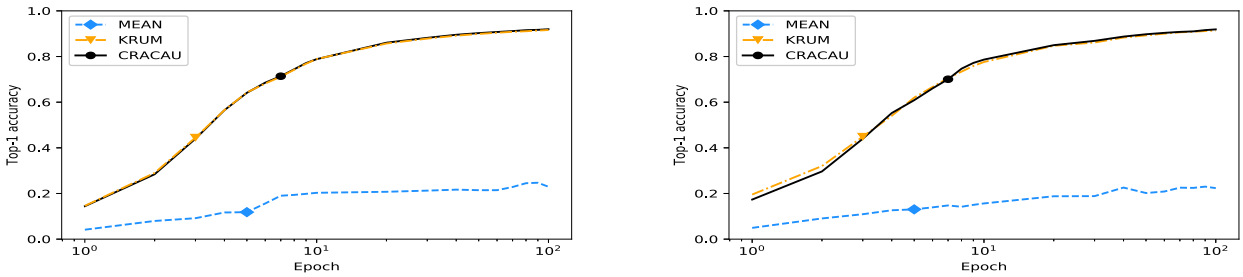


Fig. 6. Convergence of Krum [10] and CRACAU on the MNIST dataset. (a) Top-1 accuracy of i.i.d. data batches and sign-flipping attack. (b) Top-1 accuracy of disjoint local data and sign-flipping attack.

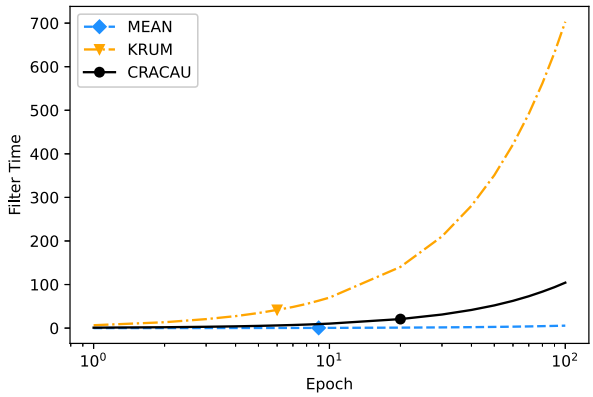


Fig. 7. Accumulated filtering time at workers in the i.i.d. MNIST dataset

of epochs) workers take to compute the gradients using three different filters, in the setting of the i.i.d. MNIST dataset. It shows that CRACAU trains much more efficiently than Krum and is comparable with MEAN, a non-fault-tolerant filter that is highly optimized with primitive support from MXNet. This was not surprising, since, as analyzed in Table I, CRACAU has a more lightweight computation at the workers. Note that Krum [10] was designed for the datacenter setting, and its proof is not on the edge computing scenario, whereas CRACAU not only is faster, but also is proven correct.

VII. Conclusion

This article presented a novel Byzantine ML algorithm that is designed for the three-level Industrial 5.0 edge computing architecture. We formally proved that the proposed algorithm, CRACAU, converges in the local Byzantine fault model. We also built a customized simulator using the MXNet framework and SUMO and conducted evaluations on two popular benchmark datasets: MNIST and CIFAR-10. Evaluations showed that CRACAU: 1) tolerates Byzantine attacks; 2) achieves comparable accuracy and computation load with baseline (MEAN) when there is no failure; and 3) is much faster than prior work Krum. CRACAU is a first step toward a Byzantine-tolerant online learning. As pointed out in Section IV-C, there are a few interesting future directions: 1) incorporating asynchronous update rules with reduced communication complexity and increased speed; 2) adopting a flexible architecture by uploading gradients to different edge servers; and 3) addressing the catastrophic forgetting issue by integrating a quadratic penalty term in the loss function.

References

[1] S. Hu, Y. Shi, A. Colombo, S. Karnouskos, and X. Li, "Cloud-edge computing for cyber-physical systems and Internet-of-Things," *IEEE Trans. Ind. Informat.*, to be published, doi: 10.1109/TII.2021.3064881.

[2] Y. Yu, S. Liu, P. Yeoh, B. Vucetic, and Y. Li, "Layerchain: A hierarchical edge-cloud blockchain for large-scale low-delay IIoT applications," *IEEE Trans. Ind. Informat.*, vol. 17, no. 7, pp. 5077–5086, Jul. 2021.

[3] A. Marchisio *et al.*, "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2019, pp. 553–559.

[4] Y. Huang, X. Ma, X. Fan, J. Liu, and W. Gong, "When deep learning meets edge computing," in *Proc. IEEE 25th Int. Conf. Netw. Protocols*, 2017, pp. 1–2.

[5] Z. Tao and Q. Li, "eSGD: Communication efficient distributed deep learning on the edge," in *Proc. USENIX Workshop Hot Topics Edge Comput.*, 2018.

[6] S. Hosseinalipour, C. G. Brinton, V. Aggarwal, H. Dai, and M. Chiang, "From federated learning to fog learning: Towards large-scale distributed machine learning in heterogeneous wireless networks," 2020, *arXiv:2006.03594*.

[7] J. Posner, L. Tseng, M. Aloqaily, and Y. Jararweh, "Federated learning in vehicular networks: Opportunities and solutions," *IEEE Netw.*, vol. 35, no. 2, pp. 152–159, Mar./Apr. 2021.

[8] *Federated Learning: Collaborative Machine Learning Without Centralized Training Data*, Apr. 2017.

[9] N. Gupta and N. H. Vaidya, "Fault-tolerance in distributed optimization: The case of redundancy," in *Proc. 39th Symp. Princ. Distrib. Comput.*, 2020, pp. 365–374.

[10] P. Blanchard, El.M.El Mhamdi, R.Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 118–128.

[11] I. A. Ridhawi, M. Aloqaily, A. Boukerche, and Y. Jararweh, "Enabling intelligent IOCV services at the edge for 5G networks and beyond," *IEEE Trans. Intell. Transp. Syst.*, to be published, doi: 10.1109/TITS.2021.3053095.

[12] O. Alfandi, S. Otoum, and Y. Jararweh, "Blockchain solution for IOT-based critical infrastructures: Byzantine fault tolerance," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp.*, 2020, pp. 1–4.

[13] C. Xie, S. Koyejo, and I. Gupta, "Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance," in *Proc. Int. Conf. Mach. Learn. Res.*, Long Beach, CA, USA, 2019, pp. 6893–6901.

[14] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," 2016, *arXiv:1602.05629*.

[15] L. Muñoz-González, K. Co, and E. Lupu, "Byzantine-robust federated machine learning through adaptive model averaging," 2019, *arXiv:1909.05125*.

[16] L. Tseng, N. H. Vaidya, and V. Bhandari, "Broadcast using certified propagation algorithm in presence of Byzantine faults," *Inf. Process. Lett.*, vol. 115, no. 4, pp. 512–514, 2015.

[17] N. H. Vaidya, L. Tseng, and G. Liang, "Iterative approximate Byzantine consensus in arbitrary directed graphs," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2012, pp. 365–374.
[18] C. A. R. Hoare, "Algorithm 65: Find," *Commun. ACM*, vol. 4, no. 7, pp. 321–322, Jul. 1961.
[19] A. Pelc and D. Peleg, "Broadcasting with locally bounded byzantine faults," *Inf. Process. Lett.*, vol. 93, no. 3, pp. 109–115, 2005.
[20] El.-M. El-Mhamdi, R. A. G. Guirguis, L. N. Hoang, and S. Rouault, "Genuinely distributed Byzantine machine learning," in *Proc. 39th Symp. Princ. Distrib. Comput.*, New York, NY, USA, 2020, pp. 355–364.
[21] Y. Chen, L. Su, and J. Xu, "Distributed statistical machine learning in adversarial settings: Byzantine gradient descent," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 2, pp. 1–25, Dec. 2017.
[22] Q. Zhang and L. Tseng, "Echo-CGC: A communication-efficient1byzantine-tolerant distributed machine learning algorithm in single-hop radio network," 2020, *arXiv:2011.07447*.
[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[24] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf

Anran Du received the B.S. degree in computer science from Boston College, Newton, MA, USA, in 2021. He is currently working toward the M.S. degree in mobile and Internet of Things engineering with Carnegie Mellon University, Pittsburgh, PA, USA.

His research interests include mobile computing, distributed machine learning, and applications in vehicular networks.

Yicheng Shen received the B.A. degree in computer science from Boston College, Newton, MA, USA, in 2021.

His research interests include machine learning, software engineering, and applications of vehicular cloud and edge computing.

Qinzi Zhang received the B.S. degree in mathematics from Boston College, Newton, MA, USA, in 2021. He is currently working toward the Ph.D. degree in computer engineering with Boston University, MA.

His research interests include distributed machine learning and machine learning theory.

Dr. Zhang was the recipient of the Undergraduate Research Fellowship from Boston College.

Lewis Tseng (Member, IEEE) received the B.S. degree in computer science, the B.S. degree in economics, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2005, 2010, and 2016, respectively.

He was with the Toyota Info Technology Center, Mountain View, CA, USA. He is currently an Assistant Professor with the Department of Computer Science, Boston College, Newton, MA, USA. His research interests include distributed computing/systems, fault-tolerant computing, blockchain-based systems, and applications in intelligent traffic systems.

Dr. Tseng is actively working on different IEEE events. He was the Chair of Workshop on Storage, Control, Networking in Dynamic Systems (SCNDS) 2017, 2018, and Workshop on Blockchain Applications and Theory (BAT) 2019 workshops. He was also the Proceeding Chair for the 2018 ACM Symposium on Principles of Distributed Computing.

Moayad Aloqaily (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of Ottawa, Ottawa, ON, Canada, in 2016.

He was an Instructor with the Department of Systems and Computer Engineering, Carleton University, Ottawa, in 2017. From 2018 to 2019, he was an Assistant Professor with the American University of Middle East, Kuwait. From 2019 to 2021, he was the Cybersecurity Program Director and an Assistant Professor with the Faculty of Engineering, Al Ain University, Abu Dhabi, United Arab Emirates. Since 2019, he has been the Managing Director of xAnalytics, Inc., Ottawa. He is currently with the Machine Learning Department, Mohamed Bin Zayed University of Artificial Intelligence (MBZUAI), UAE. His current research interests include the applications of artificial intelligence and machine learning, connected and autonomous vehicles, blockchain solutions, and sustainable energy and data management.

Dr. Aloqaily has Chaired and Cochaired many IEEE conferences and workshops. He was the Guest Editor for many journals, including *IEEE Wireless Communications Magazine*, IEEE NETWORK, and *Computer Network*. He is currently an Associate Editor for *Ad Hoc Networks*, *Cluster Computing*, *Security and Privacy*, and IEEE ACCESS. He was the Co-Editor-in-Chief of the *IEEE CommSoft TC e-Letter* in 2020. He started his own Special Interest Group on Blockchain and Application and Internet of Unmanned Aerial Networks. He is a Professional Engineer in Ontario and a Member of the Association for Computing Machinery.