Adaptive Verifiable Coded Computing: Towards Fast, Secure and Private Distributed Machine Learning

Tingting Tang
Computer Science Department
University of Southern California
Los Angeles, USA
tangting@usc.edu

Ramy E. Ali
Electrical Engineering Department
University of Southern California
Los Angeles, USA
reali@usc.edu

Hanieh Hashemi
Electrical Engineering Department
University of Southern California
Los Angeles, USA
hashemis@usc.edu

Tynan Gangwani
Electrical Engineering Department
University of Southern California
Los Angeles, USA
tgangwan@usc.edu

Salman Avestimehr
Electrical Engineering Department
University of Southern California
Los Angeles, USA
avestime@usc.edu

Murali Annavaram
Electrical Engineering Department
University of Southern California
Los Angeles, USA
annavara@usc.edu

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00067

***Abstract*—Stragglers, Byzantine workers, and data privacy are the main bottlenecks in distributed cloud computing. Some prior works proposed coded computing strategies to jointly address all three challenges. They require either a large number of workers, a significant communication cost or a significant computational complexity to tolerate Byzantine workers. Much of the overhead in prior schemes comes from the fact that they tightly couple coding for all three problems into a single framework. In this paper, we propose Adaptive Verifiable Coded Computing (AVCC) framework that decouples the Byzantine node detection challenge from the straggler tolerance. AVCC leverages coded computing just for handling stragglers and privacy, and then uses an orthogonal approach that leverages verifiable computing to mitigate Byzantine workers. Furthermore, AVCC dynamically adapts its coding scheme to trade-off straggler tolerance with Byzantine protection. We evaluate AVCC on a compute-intensive distributed logistic regression application. Our experiments show that AVCC achieves up to $4.2\times$ speedup and up to 5.1% accuracy improvement over the state-of-the-art Lagrange coded computing approach (LCC). AVCC also speeds up the conventional uncoded implementation of distributed logistic regression by up to $7.6\times$, and improves the test accuracy by up to 12.1%.**

***Index Terms*—coded computing, verifiable computing, machine learning, straggler mitigation, Byzantine robustness, privacy**

I. Introduction

Distributed machine learning using cloud resources is widely used as it allows users to offload their compute-intensive operations to run on multiple cloud servers [1]. Distributed computing, however, faces several challenges such as stragglers and compromised systems. Execution speed variations are commonly observed among compute nodes in the cloud, which can be up to an order of magnitude resulting in straggler behavior. These variations are due to the heterogeneity in server hardware, resource contention across shared virtual instances, IO delays, or even hardware faults [2]. Stragglers hamper the end-to-end system performance [3]. The second challenge is that hackers routinely compromise some machines in the cloud. These compromised nodes cause two problems. The first problem is that users' data privacy may be compromised when some hacked cloud instances collude to extract private information. In the other words, hacked workers may collude to glean information about the data that a user does not want to disclose. Second, hacked nodes may act as Byzantine nodes and return incorrect computational results to the client that may derail the training performance [4]. The goal of this work is to provide a unified efficient framework that jointly tackles stragglers, provides data privacy and eliminates Byzantine nodes.

Most prior works rely on replication to provide straggler resiliency [3], [5]–[9]. Replication, however, entails significant overhead. Since it is unknown which node may be a straggler a priori, replication strategies may pro-actively assign the same task to multiple nodes. Alternatively, reactive strategies may wait for a straggler delay to appear and then relaunch the straggling task on another node, which delays the overall execution. Coded computing based approaches are known to be more efficient when stragglers are not known a priori [10], [11]. In such approaches, a primary server encodes the data and distributes the encoded data over the workers. The workers then perform the computations over the encoded data and the desired computation can be recovered from the fastest subset of workers. For instance, the coding-theoretic approach of [10] uses a maximum distance separable (MDS) (N, K)-code for encoding the data. With (N, K) MDS coding, the data is split into K pieces and then encoded into N pieces and distributed to N workers to perform linear operations, such as matrix-vector multiplication. If a subset of K nodes ($K \leq N$) returns the result to the primary server, it can decode the full result.

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00067

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

More advanced encoding strategies mask the data with random noise with the joint aim of mitigating stragglers, ensuring data privacy as well as tackling Byzantine nodes. Specifically, Lagrange coded computing (LCC) [11] provides straggler resiliency, Byzantine robustness and privacy protection even if a subset of workers, up to a certain size, collude. LCC guarantees that the colluding workers cannot learn any information about that data in the information-theoretic sense. However, the cost of tolerating Byzantine workers with LCC is twice as the cost of tolerating stragglers. For instance, tolerating two Byzantine workers requires an additional four workers while tolerating two stragglers only requires two additional workers. As we describe in more detail later, recent works reduced the cost of tolerating Byzantine workers to be the same as the cost of tolerating stragglers at the expense of increasing the communication cost significantly [12] or a significant computation complexity [13], [14].

Inspired by the prior coding based approaches, and motivated by the large overheads faced by these approaches, we propose the *Adaptive Verifiable Coded Computing* (AVCC) framework that jointly addresses stragglers, Byzantine workers and data privacy. Unlike LCC, the cost of tolerating Byzantine workers in AVCC is the same as the cost of tolerating stragglers. AVCC achieves this improvement through a unique decoupling of the data encoding for tackling stragglers and privacy, and an orthogonal information-theoretic verifiable computing approach that uses Freivalds' algorithm [15] to detect Byzantine workers. This decoupling enables AVCC to tolerate stragglers and tackle untrusted nodes in any distributed polynomial computations. AVCC further adapts to the dynamics of the system by changing the coding strategy at runtime depending on the straggler or Byzantine prevalence.

The basic intuition behind AVCC can be provided with the following example. Consider the case when AVCC uses (N, K)-MDS coding to tolerate stragglers. MDS coding can be considered as a simplified version of LCC that can be used for linear computations. The primary server encodes the data and sends it to N workers. It then receives and decodes the fastest K out of the N worker results to compute the final result. However, AVCC 's verification process checks the integrity of the computation provided by each of the K workers (Byzantine Workers). If any one of the K workers fails the verification process, AVCC tags such a worker as a Byzantine node and discards the results provided from that node. It then has to wait for additional workers whose results can be verified before decoding the full computational output. Thus, AVCC trades-off straggler tolerance for Byzantine detection and correctly computes the result. Our experiments show that AVCC speeds up the state-of-the-art LCC implementation of distributed logistic regression by up to $4.2\times$, and improves the test accuracy by up to 5.1% accuracy. AVCC also achieves up to $7.6\times$ speedup and up to 12.1% accuracy improvement over the conventional uncoded approach of distributed logistic regression.

Organization. The rest of this paper is organized as follows. Section II provides a background about coded computing, discusses the closely-related works and our contributions. In Section III, we describe our system, the threat model, and our guarantees. Section IV introduces our adaptive verifiable coded computing framework. In Section V, we describe our experimental setup followed by extensive experiments to evaluate our method in Section VI. Finally, concluding remarks and future directions are discussed in Section VII.

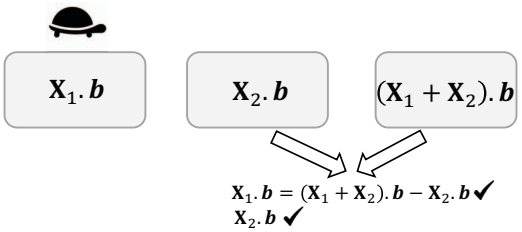


Fig. 1: An illustration of a distributed computing system using $(3, 2)$ MDS code is depicted. The goal is to compute the matrix-vector multiplication $\mathbf{X}\boldsymbol{b}$, where $\mathbf{X} = [\mathbf{X}_1^\top, \mathbf{X}_2^\top]^\top$ while tolerating one straggler. In this example, the first worker is a straggler and only the results from worker 2 and worker 3 are available.

II. Background and Related Works

In this section, we provide a brief background about coded computing, verifiable computing and the closely-related works.

A. Coded Computing

In the past, coding was used mainly to tolerate data losses during communication and data storage. However, coded computing has extended this concept to enable tolerating stragglers while performing computations. In a system with dataset $\mathbf{X} \in \mathbb{F}_q^{m\times d}$, where m is the number of samples and d is the feature size, the goal of distributed computing is to compute a multivariate polynomial $f : \mathbb{V} \to \mathbb{U}$ over $\mathbf{X} = (\mathbf{X}_1^\top, \mathbf{X}_2^\top, \cdots, \mathbf{X}_K^\top)^\top$, where $\mathbf{X}_i \in \mathbb{F}_q^{m/K\times d}$ and $\mathbb{V}$ and $\mathbb{U}$ are vector spaces of dimensions v and u, respectively, over a finite field $\mathbb{F}_q$. That is, the goal is to compute $f(\mathbf{X}_i), \forall i \in [K]$ using a distributed collection of compute nodes. We describe two approaches for encoding data: MDS which is used for linear computations, and LCC which can handle any polynomial computations.

MDS Coding. MDS coded computing is a computing paradigm that enables distributed computing on encoded data to tolerate stragglers. Fig. 1 illustrates the idea of computing a matrix-vector multiplication $\mathbf{X}\boldsymbol{b}$ using 3 workers with a $(3, 2)$ MDS code. The data matrix $\mathbf{X}$ is evenly divided into 2 submatrices $\mathbf{X}_1$ and $\mathbf{X}_2$, then encoded into 3 coded matrices $\tilde{\mathbf{X}}_1 = \mathbf{X}_1, \tilde{\mathbf{X}}_2 = \mathbf{X}_2$ and $\tilde{\mathbf{X}}_3 = \mathbf{X}_1 + \mathbf{X}_2$, and assigned to worker $1, 2$, and 3, respectively. Worker i then receives the coded matrix $\tilde{\mathbf{X}}_i$ and the vector $\boldsymbol{b}$ and starts computing $\tilde{\mathbf{X}}_i\boldsymbol{b}$, where $i \in [3]$. The final result can be recovered when the results from any 2 out of the 3 workers are received, without the need to wait for one straggler. Assume that results from worker 2 and worker 3 are received. Then $\mathbf{X}_1\boldsymbol{b}$ can be decoded by subtracting $\mathbf{X}_2\boldsymbol{b}$ from $(\mathbf{X}_1+\mathbf{X}_2)\boldsymbol{b}$, and the final result can be obtained by concatenating $\mathbf{X}_1\boldsymbol{b}$ and $\mathbf{X}_2\boldsymbol{b}$. In general, for an

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

(N, K) MDS code, the data matrix $\mathbf{X}$ is divided into K equal size sub-matrices $\mathbf{X}_1, \ldots, \mathbf{X}_K$, for $K \leq N$. Then N encoded matrices $\tilde{\mathbf{X}}_1, \ldots, \tilde{\mathbf{X}}_N$ are generated by applying (N, K)-MDS code to the sub-matrices. If a systematic MDS code is used for encoding, then $\tilde{\mathbf{X}}_i = \mathbf{X}_i$, for $1 \leq i \leq K$. Once any K out of the N results are received from the workers nodes, the master node can decode the final result using these K results.

Lagrange Coded Computing (LCC) [11]. MDS-coded computing can inject redundancy to tolerate stragglers in linear computations. LCC extends this idea for any polynomial-based computation. LCC provides a single framework to tolerate stragglers, Byzantine nodes and to protect privacy against colluding workers. LCC encodes the dataset into N coded datasets $\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2, \cdots, \tilde{\mathbf{X}}_N \in \mathbb{V}$, where N is the number of worker nodes, and the i-th node computes $f(\tilde{\mathbf{X}}_i) \in \mathbb{U}$. Specifically, LCC requires that

$$N \geq (K + T - 1) \deg f + S + 2M + 1, \tag{1}$$

to tolerate S stragglers, M Byzantine workers and to ensure privacy of the dataset against any T colluding workers, where $\deg f$ is the degree of the polynomial f. We refer to this scheme as (N, K, S, M) LCC. The overall computation results in LCC can be then recovered when at least $N - S$ nodes return their computations through a Reed-Solomon decoding approach [16]. This approach has an encoding complexity of $O(N \log^2(K) \log\log(K) v)$ and results in a decoding complexity of $O((N - S) \log^2(N - S) \log\log(N - S) u)$, where v is the output size of the encoder and u is the input size of the decoder. That is, the encoding complexity is almost linear in $O(Nv)$ and the decoding complexity is almost linear in $O((N - S)u)$. We observe from (1) that handling Byzantine workers are *twice* as costly as stragglers in LCC. Hence, Byzantine node detection is resource-intensive in LCC.

Broader use of coded computing. LCC [11] provides coded redundancy for any arbitrary multivariate polynomial computations such as general tensor algebraic functions, inner product functions, function computing outer products, and tensor computations. Polynomially coded computing [17] can tolerate stragglers in bilinear computations such as Hessian matrix computation. Recent works [18]–[22] have also demonstrated promising results for extending coded computing to beyond polynomial computations such as deep learning training and inference.

B. Verifiable Computing

Verifiable computing is an orthogonal paradigm that has been designed to ensure computational integrity [15]. The basic principle behind verifiable computing is to allow a user to verify whether a compute node has performed the assigned computation correctly. While there are a variety of approaches to verify computations, in this work we adapt the approach proposed in [15]. Consider the problem where a user is interested in computing the matrix-vector multiplication $\boldsymbol{y} = \mathbf{X}\boldsymbol{b}$ by offloading it to a worker node, where $\mathbf{X} \in \mathbb{F}_q^{m \times d}$ and $\boldsymbol{b} \in \mathbb{F}_q^d$. The user chooses a vector $\boldsymbol{r} \in \mathbb{F}_q^m$ uniformly at random and computes $\boldsymbol{s} \triangleq \boldsymbol{r}\mathbf{X}$ as a private verification key. This verification key generation is done only once. The worker node then returns the outcome $\hat{\boldsymbol{y}}$, where $\hat{\boldsymbol{y}} = \mathbf{X}\boldsymbol{b}$ if the node performed the computation correctly. The user then performs the following verification check: $\boldsymbol{r} \cdot \hat{\boldsymbol{y}} = \boldsymbol{s} \cdot \boldsymbol{b}$. If the worker node passes this verification check successfully, then the user accepts this computation result. Note that this verification step is done in only $O(m + d)$ arithmetic operations and is much faster compared to computing $\boldsymbol{y} = \mathbf{X}\boldsymbol{b}$ on the server, which incurs a complexity of $O(md)$. That is, through this step, the user performs substantially fewer operations compared to the original computation to identify any discrepancy in the computations with high probability.

C. Related Coding Strategies for Byzantine Detection

Numerous works considered the problem of tolerating Byzantine workers in distributed computing and learning settings. For instance, Draco [23] and Detox [24] introduce algorithmic redundancy that tackles the Byzantine problem in isolation without dealing with stragglers. Coding-theoretic approaches beyond LCC and Polynomial codes have been proposed recently to tolerate Byzantine nodes in distributed settings [12]–[14]. In [12], two schemes have been proposed to improve the adversarial toleration threshold of LCC based on decomposing the polynomials as a series of monomials. This decreases the effective degree of the polynomial, but increases the communication cost significantly. In [13], this problem was also considered for the Gaussian and the uniform *random* error models. Our work, however, considers the worst-case error model. More recently, a list-decoding approach with side information has been developed in [14] which uses the folding technique in algebraic coding to tolerate more Byzantine nodes. This approach, however, has a significant decoding complexity that is quadratic in the number of workers while it is almost linear in LCC.

Another line of work focused on detecting Byzantine behavior through verifiable computing. An information-theoretic verifiable computing scheme for polynomial computations was proposed in [25], [26] in the single user-server setting based on Freivalds' algorithm. Leveraging verifiable computing in machine learning has also been considered in many works such as [27]–[29]. In particular, Slalom [28] uses Trusted Execution Environment (TEE)-GPU collaboration for privacy-preserving and verifiable inference. Because of the hardware limitations of TEE and consequently its lower performance, Slalom offloads the linear operations to an untrusted GPU and uses Freivalds' algorithm for verification [15] which is less compute-intensive than the original computation. However, this scheme is designed for a single GPU system and cannot scale to the distributed system. Also, it does not support training and it does not mitigate stragglers.

D. Contributions

The question we pose in this work is whether there is a way to exploit verifiable computing in conjunction with coded computing to get the best of both worlds. That is to use coded computing to tolerate stragglers and ensure data

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

privacy, while using verifiable computing to tolerate Byzantine workers. Such a decoupled approach will lower the cost of tolerating Byzantine workers compared to LCC. We consider a general scenario in which the computation is distributed across N nodes, and propose *Adaptive Verifiable Coded Computing* (AVCC), a new framework to simultaneously mitigate stragglers, provide security against Byzantine workers and provide data privacy. Unlike LCC, AVCC only requires that

$$N \geq (K+T-1)\deg f + S + M + 1. \tag{2}$$

Compared to (1) in LCC, note that in (2) the cost of tolerating a Byzantine node is the same as that of a straggler node. Hence, instead of the $2M$ nodes required in LCC, AVCC only needs M nodes, as we demonstrate in Section IV.

The key idea of AVCC is to use separate mechanisms for mitigating straggler effects and for tackling Byzantine nodes. AVCC ensures the computational integrity by verifying the computation of each node independently using node's own compute results until it gets the minimum number of verified results required for decoding. This verification step can start as soon as the first node responds, unlike LCC which requires $N-S$ workers to respond before starting to decode. Hence, AVCC is also highly parallelizable.

In principle, AVCC can be applied to the distributed computation of any polynomial f. However, AVCC is particularly suitable for matrix-vector, matrix-matrix multiplications and in machine learning applications such as linear regression and logistic regression, as verifying such computations is highly efficient [15].

III. Problem Setting

In this section, we describe our setting, the threat model and our guarantees.

A. System Setting and Threat Model

We consider a distributed system with N nodes and a main server, denoted as master, that distributes the data among the worker nodes. Given a dataset $\mathbf{X} = [\mathbf{X}_1^\top, \mathbf{X}_2^\top, \cdots, \mathbf{X}_K^\top]^\top$, our final goal is to compute $\mathbf{Y}_i = f(\mathbf{X}_i), \forall i \in [K]$. To this end, the main server first encodes that data into N coded datasets denoted by $\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2, \cdots, \tilde{\mathbf{X}}_N$. The i-th worker then receives $\tilde{\mathbf{X}}_i$, computes $\tilde{\mathbf{Y}}_i = f(\tilde{\mathbf{X}}_i)$ and sends the result back to the main server for verification and decoding. The main server collects all computations from the non-straggling workers, first verifies them as we explain in the following sections and finally recovers the computation outcome $\mathbf{Y}_1, \cdots, \mathbf{Y}_K$ from the fastest workers that passed the verification.

We assume that the system has up to S stragglers that have higher latency compared to the other workers. While the main server is trusted, the worker nodes can be dynamically malicious. Therefore, adversaries on the workers can have full control (root access) and design any attack. As a result, at any given time, some of the worker nodes can send arbitrary results to the main server to sabotage the computation. In addition, some workers may send incorrect computations unintentionally. Specifically, we assume that up to M worker nodes can return erroneous computations with no limitation on their computational power. Finally, we assume that up to T curious workers can collude to learn information about the dataset.

B. Guarantees

Our goal is to design a scheme that provides the following guarantees.

1) S**-Resiliency**. The computation outcome must be recovered even in the presence of S stragglers.
2) M**-Security**. This means that the system can tolerate up to M workers sending erroneous computations, with no limitations on their computational capability, with an arbitrarily high probability based on verification overhead.
3) T**-Privacy**. The workers must remain oblivious to the dataset in the information-theoretic sense even if T of them collude. That is, for every set of at most T workers denoted by $\mathcal{T} \subset [N]$, we must have

$$I(\mathbf{X}; \tilde{\mathbf{X}}_{\mathcal{T}}) = 0, \tag{3}$$

where $I(.;.)$ denotes the mutual information [30] and $\tilde{\mathbf{X}}_{\mathcal{T}}$ denotes the encoded datasets at the workers in $\mathcal{T}$.

IV. Adaptive Verifiable Coded Computing (AVCC)

In this section, we present our adaptive verifiable coded computing (AVCC) framework, which consists of five key components: 1) Data encoding; 2) Verification key generation; 3) Integrity check; 4) Decoding; 5) Dynamic coding. We start with an example to illustrate the key components of AVCC.

A. Illustrating Example

We focus on the logistic regression problem to illustrate how AVCC works. Given a dataset $\mathbf{X} \in \mathbb{R}^{m \times d}$ of m data points and d features and a label vector $\boldsymbol{y} \in \mathbb{R}^m$, the goal in logistic regression training is to find the weight vector $\boldsymbol{w} \in \mathbb{R}^d$ that minimizes the cross entropy function

$$C(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}(-y_i \log \hat{y}_i - (1-y_i)\log(1-\hat{y}_i)), \tag{4}$$

where $\hat{y}_i = h(\boldsymbol{x}_i \cdot \boldsymbol{w}) \in (0,1)$ is the estimated probability of label i being equal to 1, $\boldsymbol{x}_i$ is the i-th row of $\mathbf{X}$, and $h(\cdot)$ is the sigmoid function $h(\theta) = 1/(1+e^{-\theta})$. The gradient descent algorithm solves this problem iteratively by updating the model as

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \frac{\eta}{m}\mathbf{X}^\top(h(\mathbf{X}\boldsymbol{w}^{(t)}) - \boldsymbol{y}), \tag{5}$$

where η is the learning rate and the function $h(\cdot)$ operates element-wise on the vector $\mathbf{X}\boldsymbol{w}^{(t)}$. In this example, we provide a two-round protocol as follows.[1]

In the first round, an intermediate vector $\boldsymbol{z}^{(t)} = \mathbf{X}\boldsymbol{w}^{(t)}$ is computed, which is then used to compute the predicted probability $h(\boldsymbol{z}^{(t)})$ and the prediction error vector $\boldsymbol{e}^{(t)} = h(\boldsymbol{z}^{(t)}) - \boldsymbol{y}$.

[1]The dataset and the weight vector at each round are quantized and represented over the finite field to guarantee information-theoretic privacy [31].

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

In the second round, the gradient vector $\boldsymbol{g}^{(t)} = \mathbf{X}^\top \boldsymbol{e}^{(t)}$ is computed.

We now illustrate how to compute this logistic regression task in a distributed setting in the presence of stragglers and compromised compute nodes as depicted in Fig. 2.

1) **Data Encoding.** Before starting the computation, the main server partitions the dataset $\mathbf{X}$ into K sub-matrices and encodes them using (N, K)-MDS coding. As stated earlier, MDS encoding is a special case of LCC encoding when the computations are only linear. The main server then sends the coded sub-matrix $\tilde{\mathbf{X}}_i$ to the i-th worker, where $i \in [N]$.
2) **Verification Key Generation.** The main server also computes a one-time verification key that helps in verifying the computation results returned by the worker nodes afterwards. Specifically, the main server chooses a vector $\boldsymbol{r}_i^{(1)} \in \mathbb{F}_q^{m/K}$ and a vector $\boldsymbol{r}_i^{(2)} \in \mathbb{F}_q^{d/K}$ uniformly at random for each worker $i \in [N]$.
The main server then computes the following private verification keys as in the Freivalds' algorithm [15] as follows
$$\boldsymbol{s}_i^{(1)} \triangleq \boldsymbol{r}_i^{(1)} \tilde{\mathbf{X}}_i, \tag{6}$$
$$\boldsymbol{s}_i^{(2)} \triangleq \boldsymbol{r}_i^{(2)} \tilde{\mathbf{X}}_i^\top. \tag{7}$$
The main server then keeps these private verification keys for the two rounds along with $\boldsymbol{r}_i^{(1)}$ and $\boldsymbol{r}_i^{(2)}, \forall i \in [N]$.
3) **Integrity Check.** At the first round of iteration t, when the i-th worker returns its result $\tilde{\boldsymbol{z}}_i^{(t)}$, which is $\tilde{\mathbf{X}}_i \boldsymbol{w}^{(t)}$ if this worker is not Byzantine, the main server checks the following equality
$$\boldsymbol{s}_i^{(1)} \cdot \boldsymbol{w}^{(t)} = \boldsymbol{r}_i^{(1)} \cdot \tilde{\boldsymbol{z}}_i^{(t)}. \tag{8}$$
At the second round of iteration t, when the i-th worker returns $\tilde{\boldsymbol{g}}_i^{(t)}$ which is $\tilde{\mathbf{X}}_i^\top \boldsymbol{e}^{(t)}$ if this worker is not Byzantine, the main server checks the equality
$$\boldsymbol{s}_i^{(2)} \cdot \boldsymbol{e}^{(t)} = \boldsymbol{r}_i^{(2)} \cdot \tilde{\boldsymbol{g}}_i^{(t)}. \tag{9}$$
These verification steps are done in only $O(m + d)$ arithmetic operations and are much faster compared to computing $\tilde{\boldsymbol{z}}_i^{(t)}$ and $\tilde{\boldsymbol{g}}_i^{(t)}$ which requires $O(\frac{m}{K} d)$ arithmetic operations. Through these verification steps, the main server can identify the Byzantine workers with high probability that depends on the finite field size q [25] as follows
$$\Pr[\boldsymbol{r}_i^{(1)} \cdot \tilde{\boldsymbol{z}}_i^{(t)} = \boldsymbol{r}_i^{(1)} \cdot \tilde{\boldsymbol{z}}_i'] \leq \frac{1}{q}, \tag{10}$$
$$\Pr[\boldsymbol{r}_i^{(2)} \cdot \tilde{\boldsymbol{g}}_i^{(t)} = \boldsymbol{r}_i^{(2)} \cdot \tilde{\boldsymbol{g}}_i'] \leq \frac{1}{q}, \tag{11}$$
for any $\tilde{\boldsymbol{z}}_i' \neq \tilde{\boldsymbol{z}}_i^{(t)}$ and $\tilde{\boldsymbol{g}}_i' \neq \tilde{\boldsymbol{g}}_i^{(t)}$.
4) **Decoding.** The main server decodes the results in each round using the MDS decoding process with the additional constraint that each of the K results it uses has been verified first as explained in step 3.
The decoding starts when the main server collects K verified results. Following the property of MDS coding that any $K \times K$ sub-matrix formed by any K columns of the $K \times N$ encoding matrix is invertible, the decoding algorithm is simply to multiply the result matrix formed by concatenating the returned results from K verified workers, with the inverse of the matrix formed by the K columns of the encoding matrix corresponding to the indices of the K verified workers. Thus, the main server can decode and recover the final output using K verified results, instead of just using the first K results. For Byzantine workers that fail the verification, they are effectively treated as stragglers and their results are not used for decoding.
5) **Dynamic Coding.** Ignoring a Byzantine worker's result comes at the cost of reduced straggler tolerance as the main server has to wait for an additional verified result. In the original MDS coding strategy, $N - K$ stragglers can be tolerated. If the original MDS straggler tolerance is still desired, the main server changes the coding strategy to $(N - 1, K - 1)$ to account for this Byzantine worker. In this coding strategy, each worker now performs more work but only $K - 1$ results are needed to decode. Thus, this dynamic coding approach enables the main server to trade-off redundant work with Byzantine tolerance.

Remark 1. (AVCC Improvements over LCC). AVCC has two key improvements over LCC. First, AVCC relies on LCC for straggler tolerance and ensuring privacy only but it depends on verifiable computing to identify the Byzantine workers. This verification process can be done independently for each worker node without waiting for the results of other workers. LCC, in contrast, has to wait for the results of a sufficient number of workers before identifying the Byzantine workers. This allows AVCC to use less number of workers compared to LCC. Second, AVCC uses a dynamic coding approach to automatically trade-off straggler tolerance for higher Byzantine protection and vice-versa.

B. Generalized AVCC

As explained in Section IV-A, AVCC is particularly suitable for matrix-vector and matrix-matrix multiplications as the information-theoretic verification schemes of such computations are efficient [25]. Such computations are essential in several machine learning applications such as linear regression and logistic regression. However, in principle, AVCC can be applied to any polynomial f. We now explain the encoding, the verification and the decoding of AVCC.

1) **Data Encoding.** In AVCC, the dataset $\mathbf{X} = (\mathbf{X}_1^\top, \mathbf{X}_2^\top, \cdots, \mathbf{X}_K^\top)^\top$ is encoded as in LCC, but AVCC requires less number of workers. Specifically, the encoding is as follows. First, a set of $K+T$ distinct elements denoted by $\mathcal{B} = \{\beta_1, \cdots, \beta_{K+T}\}$ are chosen from $\mathbb{F}_q$ and an encoding polynomial of degree at most $K+T-1$ is constructed such that $u(\beta_j) = \mathbf{X}_j$ for $j \in [K]$ and $u(\beta_j) = \mathbf{W}_j$ for $j \in \{K+1, \cdots, K+T\}$, where $\mathbf{W}_j$

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

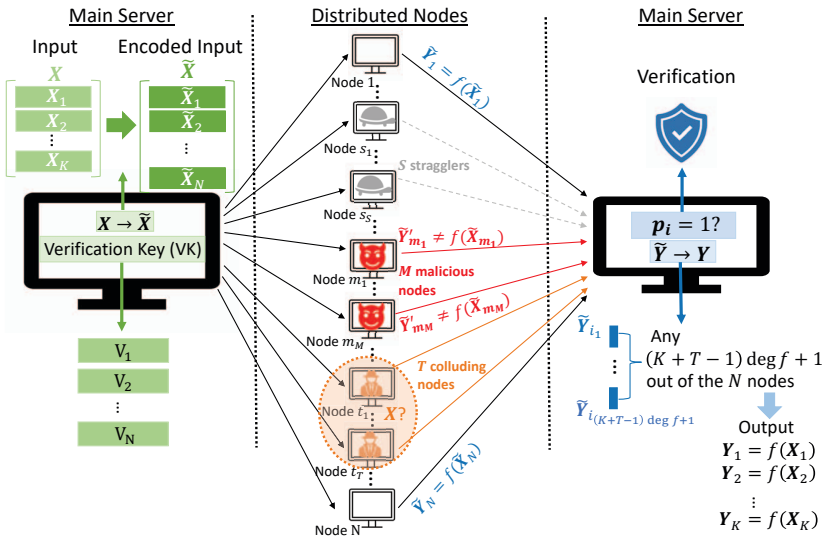


Fig. 2: An overview of the Adaptive Verifiable Coded Computing (AVCC) framework is shown. In AVCC, the main server (master) verifies the computation of each worker individually as soon as this worker sends its computation result using the initially computed verification keys. The main server then reconstructs the final output using the results of the fastest and verified workers.

is chosen uniformly at random. Such a polynomial can be constructed as follows

$$u(z) = \sum_{j=1}^{K} \mathbf{X}_j \ell_j(z) + \sum_{j=K+1}^{K+T} \mathbf{W}_j \ell_j(z), \qquad (12)$$

where $\ell_j(z)$ is the Lagrange monomial defined as follows

$$\ell_j(z) = \prod_{k \in [K+T] \setminus \{j\}} \frac{z - \beta_k}{\beta_j - \beta_k}, \qquad (13)$$

for $j \in [K+T]$. Next, another set of N distinct points denoted by $\mathcal{A} = \{\alpha_1, \cdots, \alpha_N\}$ is selected from $\mathbb{F}_q$. If $T > 0$, then the sets are selected such that $\mathcal{A} \cap \mathcal{B} = \emptyset$. The main server then sends $\tilde{\mathbf{X}}_i = u(\alpha_i)$ to the i-th worker which is required to compute $f(\tilde{\mathbf{X}}_i) = f(u(\alpha_i))$, where we note that

$$\deg f(u(z)) \leq (K+T-1)\deg f. \qquad (14)$$

The main difference between the encoding in LCC and the encoding of AVCC is that LCC requires that $N \geq (K+T-1)\deg f + S + 2M + 1$, whereas AVCC only requires that $N \geq (K+T-1)\deg f + S + M + 1$.
It is also worth noting that when $T = 0$ and $\deg f = 1$ in AVCC, we can encode the dataset using an (N, K) MDS code as illustrated in Subsection IV-A.

2) **Verification Key Generation.** The main server also computes a random private verification key $\mathbf{V}_i$ for each worker i which depends on $\tilde{\mathbf{X}}_i$ and the polynomial f.
3) **Integrity Check**. When the main server receives the result of the i-th worker, it verifies the correctness of this result using the private verification key $\mathbf{V}_i$. We denote the binary output of the verification algorithm for the i-th worker by p_i, where $p_i = 1$ if this worker passes the verification test and $p_i = 0$ otherwise.
If the i-th worker returns the correct computation result $\tilde{\mathbf{Y}}_i = f(\tilde{\mathbf{X}}_i)$, then the main server accepts the result with probability 1. Otherwise, the main server detects this malicious behavior regardless of the computational power of this worker with high probability [25]. Specifically, we have

$$\Pr(p_i = 0, \tilde{\mathbf{Y}}_i' \neq f(\tilde{\mathbf{X}}_i)) \geq 1 - o(1), \qquad (15)$$

where the term $o(1)$ is a term that vanishes as the finite field size q grows.
4) **Decoding.** Once the main server collects $(K+T-1)\deg f + 1$ verified results, it then interpolates the polynomial $f(u(z))$ and reconstructs the computation outcome by evaluating the polynomial $f(u(z))$ at β_i $\forall i \in [K]$ as $f(u(\beta_i)) = f(\mathbf{X}_i)$.
5) **Dynamic Coding.** The main server may decide to dynamically reconfigure the coded data distribution, based on the observed system behaviour in the previous iteration(s). Assume our system has N workers and initially uses (N, K) MDS coding, where $N = K+M+S+T$. We claim that the system tolerates up to S stragglers, M Byzantine workers and T colluding workers. Our strategy changes the dimension of the code and the code length dynamically based on the history. We denote the dimension of the code at time t by K_t and the number of workers in the system at time t by N_t. That is, we use (N_t, K_t) MDS code at iteration t. Suppose that at iteration t, we detect S_t stragglers and M_t malicious workers, and there are potentially T_t colluding workers in the system, where $S_t \leq S$, $M_t \leq M$ and $T_t \leq T$. We

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

define a parameter A_t that shows how many additional stragglers we can tolerate in the future iterations before we suffer from the tail latency as follows

$$A_t = N_t - M_t - S_t - K_t - T_t. \tag{16}$$

This parameter determines that the new coding scheme at iteration $t+1$ should be as follows

$$(N_{t+1}, K_{t+1}) = \begin{cases} (N_t - M_t, K_t) & \text{if } A_t \ \geq 0, \\ (N_t - M_t, K_t + A_t) & \text{if } A_t < 0. \end{cases} \tag{17}$$

That is, our strategy is as follows. If $A_t \geq 0$, we do not have to wait for any stragglers to complete the computation. However, when $A_t < 0$ this indicates that we already suffer from stragglers and we need to adapt our coding scheme with the available nodes we have. To do so, we need to reduce the dimension of the code as well as the code length. But re-encoding the data and verification keys based on the new coding scheme can be a performance bottleneck. For this reason, in the preprocessing phase before the application starts, we generate encoded data as well as verification keys of different coding configurations offline. Therefore, we have the flexibility to use them dynamically.

Similar strategy can be applied with minor modification when the system encodes using Lagrange coding. In the case of using Lagrange coding, we set A_t as follows

$$A_t = N_t - M_t - S_t - (K_t + T_t - 1)\deg f, \tag{18}$$

and the new coding scheme at iteration $t+1$ is as follows

$$(N_{t+1}, K_{t+1}) = \begin{cases} (N_t - M_t, K_t) & \text{if } A_t \ \geq 0, \\ (N_t - M_t, K_t + \lfloor \frac{A_t}{\deg f} \rfloor) & \text{if } A_t < 0. \end{cases} \tag{19}$$

Theorem 1 characterizes the set of all feasible S-resilient, M-secure, and T-private schemes that AVCC achieves.

Theorem 1. *Given a number of workers N and a dataset $\mathbf{X} = (\mathbf{X}_1^\top, \mathbf{X}_2^\top, \cdots, \mathbf{X}_K^\top)^\top$, AVCC provides an S-resilient, M-secure, and T-private scheme for computing $\{f(\mathbf{X}_i)\}_{i=1}^K$ for any polynomial f, as long as*

$$N \geq (K + T - 1)\deg f + S + M + 1. \tag{20}$$

Proof. We start by showing that AVCC is S-resilient and M-secure and then show it is T-private.

- **S-resiliency and M-security**. Since LCC uses Reed-Solomon decoder to identify the Byzantine workers, it requires $2M$ additional workers in order to tolerate M malicious workers. Unlike LCC, AVCC mitigates Byzantine workers by verifying each received worker's result independently. If the worker returns the correct computation result, then the verification algorithm accepts the result with probability 1. Otherwise, the verification algorithm rejects the erroneous result regardless of the computational power of the worker with a probability at least $1 - o(1)$, where the term $o(1)$ is a term that vanishes as the finite field size grows. Hence, AVCC requires only M additional worker results to tolerate M Byzantine workers.

  Specifically, since AVCC encodes the data the same way as LCC does as described in Section IV-B, it follows that AVCC is S-resilient and T-private as LCC. More specifically, the encoding polynomial $u(z)$ is constructed with $K + T$ distinct points and hence it is of degree at most $K + T - 1$. The computation at the worker side is to apply f on the encoded data, that is, to evaluate $f(u(z))$, and the composed polynomial $f(u(z))$ satisfies $\deg f(u(z)) \leq (K + T - 1)\deg f$.

  To recover $f(X_i)$, the master first interpolates $f(u(z))$, and then evaluates $f(u(z))$ at $\{\beta_i\}_{i\in[K]}$. To interpolate $f(u(z))$, the master needs a minimum of $\deg f(u(z))+1$ correct evaluations, that is, $\deg f(u(z))+1$ correct worker results. Since $\deg f(u(z)) \leq (K + T - 1)\deg f$ and $N \geq (K+T-1)\deg f+S+M+1$, the master is ensured to obtain $\deg f(u(z))+1$ correct worker results and then obtain all coefficients of $f(u(z))$. This is because the master has at least $(K + T - 1)\deg f + 1$ correct results without the results of the at most S stragglers and after discarding the results of at most M malicious workers. Hence, we have shown that the AVCC scheme is S-resilient and M-secure.
- **T-privacy.** We recall that AVCC uses the same encoding method as LCC. This encoding can be represented as

$$\tilde{\mathbf{X}}_i = u(\alpha_i) = (\mathbf{X}_1, \cdots, \mathbf{X}_K, \mathbf{W}_{K+1}, \cdots, \mathbf{W}_{K+T}).\mathbf{U}_i,$$

where $\mathbf{U} \in \mathbb{F}_q^{(K+T)\times N}$ is the encoding matrix,

$$\mathbf{U}_{i,j} \triangleq \prod_{k\in[K+T]\setminus\{i\}} \frac{\alpha_j - \beta_k}{\beta_i - \beta_k}$$

and $\mathbf{U}_i$ is the i-th column of $\mathbf{U}$. Then, it follows from Lemma 2 in [11] that every $T \times T$ submatrix of the bottom $T \times N$ submatrix $\mathbf{U}^{\text{bottom}}$ of the encoding matrix $\mathbf{U}$ is invertible. Thus, for every set of T workers denoted by $\mathcal{T} \subset [N]$, their encoded data $\tilde{\mathbf{X}}_\mathcal{T} = \mathbf{X}\mathbf{U}_\mathcal{T}^{\text{top}} + \mathbf{W}\mathbf{U}_\mathcal{T}^{\text{bottom}}$, where $\mathbf{W} = (\mathbf{W}_{K+1}, \cdots, \mathbf{W}_{K+T})$, and $\mathbf{U}_\mathcal{T}^{\text{top}} \in \mathbb{F}_q^{K\times T}, \mathbf{U}_\mathcal{T}^{\text{bottom}} \in \mathbb{F}_q^{T\times T}$ are the top and bottom submatrices which is formed by columns in $\mathbf{U}$ that are indexed by $\mathcal{T}$. Since $\mathbf{U}_\mathcal{T}^{\text{bottom}}$ is invertible, the added random padding $\mathbf{W}\mathbf{U}_\mathcal{T}^{\text{bottom}}$ is uniformly random. Hence, the coded data $\mathbf{X}\mathbf{U}_\mathcal{T}^{\text{top}}$ is completely masked by the uniformly random mask. This guarantees that the AVCC scheme is T-private.

□

V. Experimental setup

In this section, we describe our experimental setup. We present an empirical study of the performance of AVCC compared to LCC as well as the uncoded baseline. Our focus is on training a logistic regression model for image classification, while the computation load is distributed on multiple nodes on

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

the DCOMP testbed platform [32]. In our experimental setup, we focus on the case where $T = 0$.

We train the logistic regression model described in Section IV-A for binary image classification on the GISETTE dataset [33] to experimentally examine two aspects: the performance gain of AVCC in terms of the training time and accuracy, and the trade-off between various dynamic coding strategies. The size of the GISETTE dataset is $(m, d) = (6000, 5000)$. Our experiments are deployed on a cluster of 13 Minnow instances on a DCOMP testbed, where 1 node serves as the main server and the remaining $N = 12$ nodes are worker nodes. Each Minnow node is equipped with a quad-core Intel Atom-E processor, 2GB of RAM and two 1 GbE network interfaces.

We use $(N, K, S, M) = (12, 9, 1, 1)$ configuration for the LCC baseline in the experiments. For AVCC we use $(N, K, S + M) = (12, 9, 3)$. Recall that the encoding process of AVCC only relies on N, K and hence S, M play no role in the encoding.

We also implement an uncoded baseline which has no redundancy and only 9 out of the 12 workers participate in the computation, each of them storing and processing $\frac{1}{9}$ fraction of uncoded rows from the input matrix. The main server waits for all 9 workers to return, and does not need to perform decoding to recover the result.

Quantization and Parameter Selection. Since the Lagrange coding and the integrity check technique work over a finite field $\mathbb{F}_q$, but the inputs and the weights for the model training are often defined in real domain, AVCC needs to quantize the inputs and model weights to integers as

$$x_r = \text{round}(2^l \cdot x), \tag{21}$$

where x represents a floating point number and l is the quantization parameter (number of precision bits). We then embed these integers in the finite field $\mathbb{F}_q$ of integers modulo a prime q. If the integer is negative, we represent it in the finite field using the two's complement representation. When the computation results of the workers are received by the main server, q is subtracted from all the elements larger than $\frac{q-1}{2}$ to restore the negative numbers. The results are then scaled by 2^{-l} to convert them back to real numbers. There are many prior works that use quantization schemes in training machine learning models [34]–[36] without noticeable loss in accuracy.

Matrix multiplication and vector inner product operations are performed in the logistic regression application. Hence, it is necessary to select the field size of each operand to be such that the worst-case computation output still fits within the finite field to avoid a wrap-around which may lead to an overflow error. The Minnow nodes use a 64-bit implementation, and the number of features in the GISETTE dataset is $d = 5000$. Hence, the worst-case operation must satisfy $d(q-1)^2 \leq 2^{63} - 1$. As such, we select the finite field size to be $q = 2^{25} - 39$ (the largest prime number with 25 bits) to satisfy this limitation. In our experimental setup, the GISETTE dataset values are all non-negative integers and fit within the selected finite field. Hence, no quantization is necessary. For the model weights, we optimize the quantization parameter to $l = 5$ by taking into account the trade-off between the rounding and the overflow error. Note that the bias is implemented as part of the weights, so it shares the same quantization parameter as the weights.

Byzantine Attack Models. We consider the following Byzantine attack models that are widely considered in previous works [13], [37], [38].

- **Reversed Value Attack.** In this attack, the Byzantine workers that were supposed to send $\boldsymbol{z} \in \mathbb{F}_q^{m/K}$ to the main server instead send $-c\boldsymbol{z}$, for some $c > 0$. We set $c = 1$ in our experiment.
- **Constant Byzantine Attack.** In this model, the Byzantine workers always send a constant vector to the main server with dimension equal to that of the true result.

End-to-end Convergence Performance. We evaluate the end-to-end convergence performance of AVCC, under two setups and the two attack models, and compare it to LCC and the uncoded approaches. The LCC baseline is designed for $(S = 1, M = 1)$, so it requires $K+S+2M = 12$ workers. AVCC, LCC and the uncoded baseline are all trained for 50 iterations. For AVCC we use two different setups while staying within the constraints that $12 \geq 9 + S + M$ as follows.

1) $S = 1, M = 2$. In this setup, up to 2 Byzantine nodes may be tolerated but at the expense of reducing the straggler tolerance from a maximum of 3 nodes to only 1 node.
2) $S = 2, M = 1$. In the second setup, we reduce the Byzantine tolerance to 1 node, while reducing straggler tolerance to 2 nodes.

VI. Experimental Results

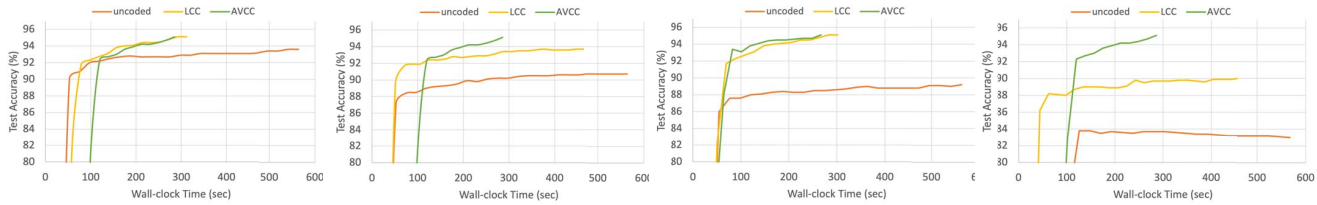
We now present extensive experimental results showing the performance gain of AVCC over LCC and the uncoded baselines.

Accuracy. We first consider the reverse value attack. This is a weak attack, as the small values produced during the matrix-vector operations when added or subtracted do not derail the overall training convergence dramatically.

Fig. 3(a) shows the test accuracy under the reverse value attack when there is only one Byzantine node in the system. In this case, since LCC is designed for $(S = 1, M = 1)$, it converges to the same accuracy as AVCC under $(S = 2, M = 1)$ setup, but AVCC reaches this accuracy level faster than LCC, because LCC by design needs to wait for 11 worker results before it can start decoding, whereas AVCC requires only 9 verified results. Under $(S = 2, M = 1)$ setup, LCC is bound to suffer tail latency from the faster of the two stragglers, while AVCC in the worst-case can start decoding after verifying results from the 9 honest workers and rejecting the Byzantine worker's result, without the need to wait for any of the two stragglers.

Fig. 3(b) shows how the test accuracy varies with training time under the reverse value attack when there are two Byzantine nodes in the system. Recall that LCC is able to handle only one Byzantine node with $N = 12$, $K = 9$ and $S = 1$ by design. In order to handle two Byzantine nodes

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.



(a) Reverse $S = 2, M = 1$ (b) Reverse $S = 1, M = 2$ (c) Constant $S = 2, M = 1$ (d) Constant $S = 1, M = 2$

Fig. 3: Convergence performance of AVCC, LCC and uncoded methods with (a) $S = 2, M = 1$ under reverse value attack, (b) $S = 1, M = 2$ under reverse value attack, (c) $S = 2, M = 1$ under constant attack and (d) $S = 1, M = 2$ under constant attack.

(while keeping $S = 1$), LCC should either increase N to be 14 or decrease K to be 7. In the first case, LCC would use more worker nodes (two worker nodes) and correspondingly is more expensive to implement. In the second case, LCC has to assign a lot more work to each worker node, thereby increasing the overall execution latency of each worker. However, AVCC can dynamically adapt to the changing straggler and Byzantine conditions and automatically adapt to the scenario where there are more Byzantines by trading-off straggler tolerance. Hence, with the $(S = 1, M = 2)$ setup, AVCC converges to higher accuracy than LCC. Note that in both scenarios in Fig. 3(a) and Fig. 3 (b), the uncoded scheme does not converge to the same accuracy as AVCC since it is unable to detect and isolate the Byzantine workers. Hence, the incorrect computations of the Byzantine workers degrade the overall accuracy.

Fig. 3(c) shows how the test accuracy varies with training time under the constant attack with one Byzantine node. Compared to the reverse value attack, the constant attack is a stronger attack as the malicious behaviour forces all values to a constant value and often causes a considerable accuracy degradation. LCC is able to detect and isolate the one Byzantine node and hence it reaches an accuracy that is on par with AVCC . AVCC is able to achieve the accuracy at least 10% faster. However, AVCC shines when the number of Byzantine nodes increases past one.

With the $(S = 1, M = 2)$ setup shown in Fig. 3(d) with the constant attack, LCC converges to 90% accuracy whereas the uncoded scheme converges to a lower accuracy of 83%. This result is expected for the uncoded scheme, because with $M = 2$ there are two Byzantine nodes in the system that drag the accuracy down when compared to a single Byzantine node setup. As for LCC, it is designed to tolerate $(S = 1, M = 1)$, which protects LCC from one malicious node only. Hence, the accuracy of LCC is lowered by the additional malicious node existing in the system, and converges only to an accuracy of 90%. As explained earlier, if the number of Byzantine nodes increases LCC would need either more workers or assign more work per each worker node, both of which are undesirable.

Training Time. Applying AVCC leads to significant end-to-end speedups as shown in Table I. In particular, AVCC leads to $4.2\times$ speedup gain over LCC and more than $5\times$ speedup over the uncoded scheme under the constant attack. Under the reverse value attack, AVCC achieves up to $2.7\times$ speedup over LCC and more than $5.1\times$ speedup over the uncoded approach.

TABLE I: Speedups of AVCC over LCC and the uncoded scheme under various settings.

Setting	LCC	Uncoded
Reverse value attack $S = 1, M = 2$	$2.66\times$	$5.13\times$
Reverse value attack $S = 2, M = 1$	$1.09\times$	$3.22\times$
Constant attack $S = 1, M = 2$	$4.17\times$	$5.41\times$
Constant attack $S = 2, M = 1$	$1.13\times$	$7.64\times$

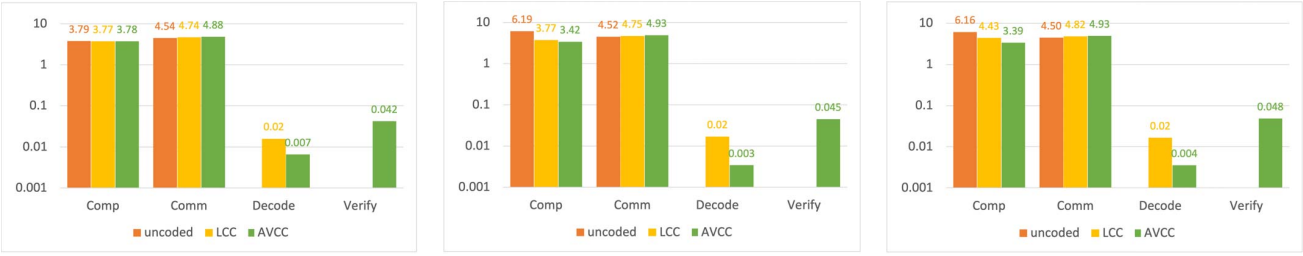
Per Iteration Cost of AVCC. The per iteration cost of applying AVCC to logistic regression using GISETTE dataset is shown in Fig. 4; note the logarithmic scale on the Y-axis. We breakdown the iteration cost into four categories as follows.

1) **Compute Time**. This is the worst-case latency for performing the matrix operations at any worker node.
2) **Communication Time**. This accounts for the time to send and receive data between the workers and the main server.
3) **Verification Time**. This is the time to verify the results. Note that the cost of encoding and key generation are one-time costs, which get amortized over multiple iterations of the model training or inference.
4) **Decoding Time**. This is the decoding time at the main server after the verification.

As shown in Fig. 4(a), in a straggler-free and Byzantine-free environment, the decoding and verification time of AVCC incurs extra latency. But when there are stragglers in the cluster, the decoding and verification overhead in AVCC is dwarfed by the straggler latency as shown in Fig. 4(b) and Fig. 4(c). Note that LCC has no separate verification cost since that process is coupled with the decoding process. The presence of stragglers causes the uncoded execution time to increase substantially. However, both LCC and AVCC are able to tolerate stragglers and Byzantine nodes. and AVCC achieves superior test accuracy even with higher Byzantine node counts.

Dynamic Coding. Recall that AVCC has the ability to dynamically change the coding strategy if straggler or Byzantine nodes persist in the system by re-encoding the data. We evaluate the benefits provided by re-encoding the data over an approach that performs AVCC functions but without re-encoding the data. We call that approach Static VCC.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.



(a) $S = 0, M = 0$

(b) $S = 1, M = 2$ (LCC test accuracy 93.7%, AVCC test accuracy 95.1%)

(c) $S = 2, M = 1$ (Test accuracy of both LCC and AVCC 95.1%)

Fig. 4: Per iteration runtime analysis of AVCC, LCC and uncoded baseline under different numbers of stragglers and Byzantine nodes. Results under constant attack are similar to that under reverse value attack and thus only results under reverse value attack are shown in (b) and (c).

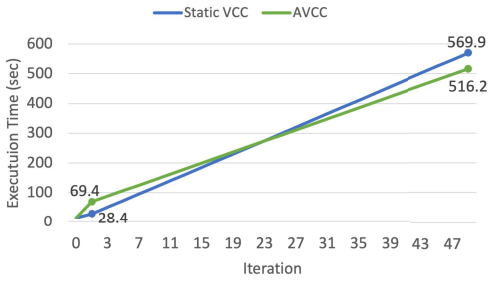


Fig. 5: AVCC and Static VCC execution time comparison

Static VCC is a constrained version of AVCC, where the verification mechanism is still available to mitigate Byzantine nodes, but the dynamic coding is removed so that the coding scheme will not change throughout the execution. The primary disadvantage of dynamic re-coding is that newly encoded data must be re-sent to the workers once such a decision is made by AVCC.

Fig. 5 quantifies the benefit of AVCC over Static VCC in an exemplary scenario. In this scenario we start with the initial coding scheme of $(N = 12, K = 9, S = 2, M = 1)$. That means we can tolerate one Byzantine node and two stragglers. At the start of iteration 1, the system encounters three stragglers and one Byzantine node. At this juncture, AVCC can eliminate the one Byzantine node from the group of workers, but it also recognizes that the coding scheme is no longer able to handle 3 stragglers. AVCC then re-encodes the data using $(N = 11, K = 8, S = 3, M = 0)$ setting. Static VCC, on the other hand, does not re-encode the data. AVCC incurs a one-time cost of about 41 seconds at the end of iteration 1, because it sends the encoded matrices with the updated coding scheme $(11, 8)$ to the workers. In spite of this re-encoding cost, at the end of the 50 iterations, AVCC saves about 54 seconds in the overall execution time, compared to Static VCC. This scenario exemplifies the benefits of the adaptive nature of AVCC. It is also worth noting that the one-time cost of re-encoding and data transfer overhead can be mitigated in different ways. For instance, it is possible for the main server to generate a priori multiple versions of the encoded data and send those versions to the workers, where each version is encoded with a different coding scheme. An alternative scheme would allow the main server to reactively encode the data off the critical path.

VII. Conclusion

In this work, we presented AVCC, a framework for resilient, robust, and private distributed machine learning via coded computing with dynamic coding and verifiable computing. AVCC is robust to up to M malicious nodes, tolerates up to S stragglers, and provides privacy against up to T colluding nodes, while being several times faster than the state-of-the-art LCC approach. Unlike prior coded computing approaches, AVCC decouples the computational integrity check from the straggler tolerance thereby reducing the cost of Byzantine tolerance.

AVCC opens the door for several interesting future directions. The encoding, decoding, and data distribution process is conducted by a trusted central server. The question to pose next is whether this central server could also be removed from our trust base. We believe that using a trusted execution environment such as an Intel SGX [28], [39], [40] equipped cloud server, one can move the vulnerable computations such as encoding and decoding to a hardware assisted secure enclave. Second, deep neural networks have non-linear computations that are difficult to decode when such computations are applied to encoded data. One potential option is to approximate such non-linearities using polynomials or rational functions [21], [22], [29], [41]. This approximation comes at the cost of accuracy loss. However, it can defend against Byzantine workers attacks.

Acknowledgment

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053 and FA8750-19-2-1005, ARO award W911NF1810400, NSF grants CCF-1703575, CCF-1763673, and MLWINS-2002874, ONR Award No. N00014-16-1-2189, and a gift from Intel/Avast/Borsetta via the PrivateAI institute, a gift from Cisco, and a gift from Qualcomm. The views,

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.

opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] X. Xu, "From cloud computing to cloud manufacturing," *Robotics and computer-integrated manufacturing*, vol. 28, no. 1, pp. 75–86, 2012.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 265–278.

[3] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[4] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 118–128.

[5] "Apache Hadoop," 2021, http://hadoop.apache.org/, last accessed on 05/01/2021.

[6] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 513–527.

[7] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" *IEEE Transactions on Communications*, vol. 64, no. 2, pp. 715–722, 2015.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 185–198.

[9] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia, "Reducing latency via redundant requests: Exact analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 347–360, 2015.

[10] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2017.

[11] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 1215–1225.

[12] C.-S. Yang and A. S. Avestimehr, "Coded computing for secure boolean computations," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 1, pp. 326–337, 2021.

[13] A. M. Subramaniam, A. Heidarzadeh, and K. R. Narayanan, "Collaborative decoding of polynomial codes for distributed computation," in *2019 IEEE Information Theory Workshop (ITW)*. IEEE, 2019, pp. 1–5.

[14] M. Soleymani, R. E. Ali, H. Mahdavifar, and A. S. Avestimehr, "List-decodable coded computing: Breaking the adversarial toleration barrier," *arXiv preprint arXiv:2101.11653*, 2021.

[15] R. Freivalds, "Probabilistic machines can use less running time," in *IFIP Congress*, 1977.

[16] E. R. Berlekamp, "Non-binary bch decoding," North Carolina State University. Dept. of Statistics, Tech. Rep., 1966.

[17] Q. Yu, M. A. Maddah-Ali, and S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," in *NIPS*, 2017.

[18] J. So, B. Guler, and S. Avestimehr, "A scalable approach for privacy-preserving collaborative machine learning," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[19] H. V. K. G. Narra, Z. Lin, G. Ananthanarayanan, S. Avestimehr, and M. Annavaram, "Collage inference: Using coded redundancy for lowering latency variation in distributed image classification systems," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 453–463.

[20] J. Kosaian, K. Rashmi, and S. Venkataraman, "Parity models: erasure-coded resilience for prediction serving systems," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 30–46.

[21] T. Jahani-Nezhad and M. A. Maddah-Ali, "Berrut approximated coded computing: Straggler resistance beyond polynomial computing," *arXiv preprint arXiv:2009.08327*, 2020.

[22] M. Soleymani, R. E. Ali, H. Mahdavifar, and A. S. Avestimehr, "Approxifer: A model-agnostic approach to resilient and robust prediction serving systems," *arXiv preprint arXiv:2109.09868*, 2021.

[23] L. Chen, H. Wang, Z. Charles, and D. Papailiopoulos, "Draco: Byzantine-resilient distributed training via redundant gradients," in *International Conference on Machine Learning*. PMLR, 2018, pp. 903–912.

[24] S. Rajput, H. Wang, Z. Charles, and D. Papailiopoulos, "Detox: A redundancy-based framework for faster and more robust gradient aggregation," *arXiv preprint arXiv:1907.12205*, 2019.

[25] S. Sahraei and A. S. Avestimehr, "Interpol: Information theoretically verifiable polynomial evaluation," in *IEEE International Symposium on Information Theory (ISIT)*, 2019, pp. 1112–1116.

[26] S. Sahraei, S. Avestimehr, and R. E. Ali, "Infocommit: Information-theoretic polynomial commitment and verification," *arXiv preprint arXiv:2002.00559*, 2020.

[27] Z. Ghodsi, T. Gu, and S. Garg, "Safetynets: Verifiable execution of deep neural networks on an untrusted cloud," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[28] F. Tramèr and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," *ArXiv*, vol. abs/1806.03287, 2019.

[29] R. E. Ali, J. So, and A. S. Avestimehr, "On polynomial approximations for privacy-preserving and verifiable relu networks," *arXiv preprint arXiv:2011.05530*, 2020.

[30] E. T. Jaynes, "Information theory and statistical mechanics. ii," *Physical review*, vol. 108, no. 2, p. 171, 1957.

[31] J. So, B. Güler, and A. S. Avestimehr, "Codedprivateml: A fast and privacy-preserving framework for distributed machine learning," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 1, pp. 441–451, 2021.

[32] R. Goodfellow, S. Schwab, E. Kline, L. Thurlow, and G. Lawler, "The dcomp testbed," in *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/cset19/presentation/goodfellow

[33] I. Guyon, S. Gunn, A. B. Hur, and G. Dror, "Result analysis of the nips 2003 feature selection challenge," in *Proceedings of the 17th International Conference on Neural Information Processing Systems*, ser. NIPS'04. Cambridge, MA, USA: MIT Press, 2004, p. 545–552.

[34] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[35] J. So, B. Guler, and A. S. Avestimehr, "Byzantine-resilient secure federated learning," *arXiv preprint arXiv:2007.11115*, 2020.

[36] H. Hashemi, Y. Wang, and M. Annavaram, "Darknight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 212–224.

[37] S. Prakash and A. S. Avestimehr, "Mitigating byzantine attacks in federated learning," *arXiv preprint arXiv:2010.07541*, 2020.

[38] H. Hashemi, Y. Wang, C. Guo, and M. Annavaram, "Byzantine-robust and privacy-preserving framework for fedml," *arXiv preprint arXiv:2105.02295*, 2021.

[39] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[40] Y. Niu, R. E. Ali, and S. Avestimehr, "Asymml: An asymmetric decomposition framework for privacy-preserving dnn training and inference," *arXiv preprint arXiv:2110.01229*, 2021.

[41] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2505–2522.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 28,2023 at 11:53:27 UTC from IEEE Xplore. Restrictions apply.