arXiv:2107.12958v1 [cs.DC] 27 Jul 2021

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

TINGTING TANG, RAMY E. ALI, HANIEH HASHEMI, TYNAN GANGWANI, SALMAN AVESTIMEHR, and MURALI ANNAVARAM, University of Southern California, USA

Stragglers, Byzantine workers, and data privacy are the main bottlenecks in distributed cloud computing. Several 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 malicious 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 work, we propose **V**erifiable **C**oded **C**omputing (**VCC**) framework that decouples Byzantine node detection challenge from the straggler tolerance. VCC leverages coded computing just for handling stragglers and privacy, and then uses an orthogonal approach of verifiable computing to tackle Byzantine nodes. Furthermore, VCC dynamically adapts its coding scheme to tradeoff straggler tolerance with Byzantine protection and vice-versa. We evaluate VCC on compute intensive distributed logistic regression application. Our experiments show that VCC speeds up the conventional uncoded implementation of distributed logistic regression by $3.2\times - 6.9\times$, and also improves the test accuracy by up to 12.6%.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; *Availability*; *Redundancy*; • **Security and privacy** → **Distributed systems security**.

Additional Key Words and Phrases: coded computing, verifiable computing, machine learning, straggler mitigation, byzantine robustness

1 INTRODUCTION

Distributed machine learning using cloud resources is being widely used as it allows users to offload their compute-intensive algebraic operations to run on multiple cloud servers [27]. Distributed computing, however, must tackle several challenges such as stragglers and compromised systems. Execution speed variations are commonly observed among compute nodes in the cloud. Sometimes the variations 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 [3]. Stragglers hamper the end-to-end system performance [7]. The second challenge is that hackers routinely compromise a subset of machines in the cloud. These compromised nodes cause two problems. User's data privacy may be compromised when a subset of hacked cloud instances collude to extract private information. In the other words, a subset of workers may collaborate to glean information about the data. Furthermore, hacked nodes may act as Byzantine nodes and return incorrect computational results to the client that may derail training convergence or accuracy [5]. The goal of this work is to provide a unified framework that tackles stragglers, provides data privacy and eliminates Byzantine nodes.

Prior works relied on replication to provide straggler resiliency [1, 2, 7, 9, 19, 25]. Replication, however, entails significant overhead. First, it is unknown which node may be a straggler a priori. Hence, replication strategies may pro-actively assign the same task to multiple nodes. Alternatively, the re-active strategies may wait for a straggler delay and then relaunch the same task which may cause delays in execution. Coded computing based approaches, however, are known to be more efficient when stragglers are not known a priori [14, 29]. In such approaches, a master server encodes the data and distributes the encoded data over the workers. The workers then do 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 [14] uses a maximum distance separable (MDS) (N, K)-code for encoding the data.

With MDS coding the data is split into K pieces, 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 master, the master can decode the full result.

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) [29] provides Byzantine tolerance and privacy protection even if a subset of workers, up to a certain size, collude. LCC guarantees that 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, in a distributed support vector machine (SVM) training, tolerating two Byzantine nodes requires an additional four worker nodes. 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 [28] or a significant computation complexity [23, 24].

Inspired by these prior coding approaches, and motivated by the large overheads faced by these approaches, we propose the verifiable coded computing (VCC) framework that jointly addresses stragglers, Byzantine workers and data privacy. Unlike LCC, the cost of tolerating Byzantine workers in VCC is the same as the cost of tolerating stragglers. VCC achieves this improvement through a unique decoupling of the data encoding for tackling stragglers and privacy, and an orthogonal information-theoretic verifiable computing approach [18] that uses Freivald's algorithm to detect Byzantine workers. This decoupling enables VCC to tolerate stragglers and tackle untrusted nodes in any distributed polynomial computations. For instance, when VCC is built on top of MDS coding it receives and decodes the fastest K worker results to compute the final result. However, concurrently its verification process checks the integrity of the computation provided by each of the K workers. If any one of the K workers fails the verification process, VCC 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, VCC trades-off straggler tolerance for Byzantine detection and correctly computes the result. VCC further adapts to system dynamics by changing the coding strategy at runtime depending on the straggler or Byzantine prevalence. Our experiments show that VCC speeds up the conventional uncoded implementation of distributed logistic regression by 3.2×-6.9×, and also improves the test accuracy by up to 12.6% by detecting and discarding Byzantine node computations.

Organization. The rest of this paper is organized as follows. Section 2 provides a background about coded computing, discusses the closely-related works and our contributions. In Section 3, we describe our system, the threat model, and our guarantees. Section 4 introduces our verifiable coded computing framework. In Section 5, we provide extensive experiments to evaluate our method. Finally, concluding remarks and future directions are discussed in Section 6.

2 BACKGROUND AND RELATED WORKS

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

2.1 Coded Computing

In the past, coding was used to tolerate data losses during communication and data storage. However, coded computing is a concept that enables us to compute on coded data. 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 $f : \mathbb{V} \rightarrow \mathbb{U}$

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.

MDS Coding. MDS coded computing is a computing paradigm that enables distributed computing on encoded data. Fig. 1 illustrates the idea of computing $\mathbf{X}\boldsymbol{b}$ using 3 worker with a $(3, 2)$ MDS code. The data matrix $\mathbf{X}$ is evenly divided into 2 sub-matrices $\mathbf{X}_1$ 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. After the 3 nodes receives the vector $\boldsymbol{b}$ and start computation, the final result can be recovered when results from any 2 out of the 3 worker nodes are received, without the need to wait for the straggler node. Assume that results from node 2 and node 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 $\mathbf{X}_2\boldsymbol{b}$.

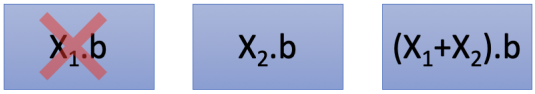

Fig. 1. Illustration of a $(3, 2)$ MDS code.

In general, for an (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 each element of 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 the K results.

Lagrange coded computing. MDS-coded computing can inject redundancy to tolerate stragglers in linear computations. Coded computing is applicable to a wider range of compute intensive algorithms, going beyond linear computations. The Lagrange coded computing (LCC) framework encodes the dataset into N coded datasets $\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2, \cdots, \tilde{\mathbf{X}}_N$, where N is the number of worker nodes, and the i-th node computes $f(\tilde{\mathbf{X}}_i)$. Specifically, LCC requires that

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

to tolerate S stragglers and M malicious nodes and to ensure privacy of the dataset against any T colluding workers, where $\deg f$ is the degree of the polynomial f. 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 [4]. This approach has an encoding complexity of $O(N \log^2(K) \log\log(K)v)$ and results in a decoding complexity of $O((N-S) \log^2(N-S) \log\log(N-S)u)$. That is, the encoding complexity is almost linear in Nv and the decoding complexity is almost linear in $(N - S)u$. LCC provides a single coding strategy to tolerate stragglers, Byzantine nodes and to protect privacy against colluding workers. We observe from Equation (1) that handling malicious nodes are *twice* as costly as stragglers in LCC. Hence, Byzantine node detection is resource intensive in LCC.

Broader use of coded computing. Lagrange coded computing [29] 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. Polynomial coded computing [30] can tolerate stragglers in bilinear computations such as Hessian matrix computation. Recent works [13, 15] demonstrate promising results by extending coded computing to non-linear applications such as deep learning inference. Finally, coded computing has also been leveraged recently for secure and privacy preserving distributed machine learning [22].

2.2 Verifiable Computing

Verifiable computing is an orthogonal paradigm that has been designed to ensure computational integrity [8]. 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 [8]. Consider the problem where a user is interested in computing the matrix-vector multiplication $\mathbf{X}\boldsymbol{b}$ by offloading it to a server. The user chooses a vector $\boldsymbol{r} \in \mathbb{F}_q^m$ uniformly at random. The user first computes $\boldsymbol{s} \triangleq \boldsymbol{r}\mathbf{X}$ as a private verification key. This verification key generation is done only once. The server computes $\boldsymbol{y} = \mathbf{X}\boldsymbol{b}$. The user then performs the following verification check: $\boldsymbol{r} \cdot \boldsymbol{y} = \boldsymbol{s} \cdot \boldsymbol{b}$. 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. Through this step, the user performs substantially fewer operations compared to the original computation to identify any discrepancy in the computations.

2.3 Related Coding Strategies for Byzantine Detection

Numerous works have considered the problem of tolerating Byzantine workers in distributed computing and learning settings. For instance, Draco [6] and Detox [17] introduce algorithmic redundancy to mitigate Byzantine workers in gradient-based computations in distributed machine learning systems. However, they neither address straggler mitigation nor data privacy. Coding-theoretic approaches beyond LCC and Polynomial codes have been proposed recently to tolerate Byzantine nodes in distributed settings [23, 24, 28]. In [28], two schemes have been proposed to improve the adversarial toleration threshold of LCC. First, a coded data logarithm approach has been developed in which the polynomial is decomposed into its monomials such that each monomial is computed separately. Specifically, applying the logarithm to the absolute value of each monomial leads to a degree-1 multivariate polynomial. Hence, this scheme can tolerate up to $M = \lfloor \frac{N-K}{2} \rfloor$ Byzantine workers when $S = 0$ and $T = 0$. This approach, however, has a decoding complexity of $O(r(f)N \log^2 N \log\log N)$ where $r(f)$ if the number of monomials of the polynomial to be computed. If the number of monomials is large, this scheme becomes much more inefficient than LCC. Second, a coded data augmentation scheme has been proposed which pre-computes values of monomials of a certain order before sending them to worker nodes. This decreases the effective degree of the polynomial, but incurs extra cost at the master node due to these increased computations and increases the amount of data sent to worker nodes which increases the communication cost.

In [24], the fact that polynomial codes [30] can be viewed as interleaved generalized Reed-Solomon codes is leveraged to correct $M \leq N - K - 1$ adversarial nodes. However, this work only considers random error models such as the Gaussian error model and the uniform random error for finite fields model. In this work, up to $M = \frac{L}{L+1}(N - K)$ Byzantine workers can be tolerated for any positive integer L. This corresponds to tolerating $M = N - K - 1$ Byzantine workers when $L \geq N - K - 1$ with a significant complexity of $O(LM^2 + N)$.

More recently, a list decoding approach with side information has been developed in [23] which uses the folding technique in algebraic coding to tolerate more Byzantine nodes. In this approach, the master node computes some extra evaluations of the desired polynomial and uses this as a side information to prune the output of the list decoder and recover the computation outcome. Specifically, two approaches have been developed in this work. First, a deterministic scheme in which the computation of the side information is done after the results are returned from the nodes. This scheme, however, increases the overall latency of decoding. Second, a probabilistic scheme in which the side information is computed in parallel to hide this latency behind the compute latency. These algorithms can tolerate

$M = \lfloor (1-\epsilon)(N-S) - (K+T)\deg(f) - 1 \rfloor$ malicious nodes for arbitrary ϵ, with a folding parameter $m = O(\frac{1}{\epsilon^2})$, as opposed to only tolerating $M = \lfloor \frac{(N-S)-(K+T-1)\deg(f)-1}{2} \rfloor$ in LCC. Thus, such schemes can tolerate almost twice adversarial nodes as LCC, albeit with a higher computational complexity of $O((N^2m^2 + m^2K^2s)v)$, where s is the number of extra evaluations computed as a side information.

Another line of work focused on detecting Byzantine behavior through verifiable computing. An information-theoretic verifiable evaluation scheme known as INTERPOL has been proposed in [18]. In this work, a user delegates the task of evaluating a polynomial f to an untrusted server. When the server returns the evaluation, the user must be able to verify the correctness of this evaluation. Specifically, the user must be able to discover any malicious behavior from the server with high probability even if the server is computationally unbounded. The key idea of INTERPOL is that it decomposes the evaluation of f into a matrix-vector multiplication and a vector-vector multiplication. The server computes the expensive matrix-vector multiplication, whereas the user verifies the result of this matrix-vector multiplication as in the Freivald's algorithm [8] and then computes the simple vector-vector multiplication to get the desired evaluation. While INTERPOL has focused on univariate polynomials in a single user-server setting, it has also been extended to the multi-party setting. In this setting, a network of untrusted nodes is considered with the goal of computing a polynomial f collaboratively. To do so, the polynomial is encoded using a Reed-Solomon code and verifiable computing is leveraged to identify the Byzantine nodes. Finally, INTERPOL also has been extended to the multi-variate polynomial case. However, the proposed scheme in this case is not efficient. Specifically, for an ℓ-variate polynomial of degree $n-1$ in each variable, the complexity of the verification is is $O(\ell^{n/2})$ and the server's complexity is $O(\ell^n)$.

The idea of leveraging verifiable computing has been also explored before in other settings. Specifically, using verifiable computing in a distributed computing setting was considered in [31]. In this work, two constructions were proposed based on Shamir's secret sharing algorithm [20] to ensure data privacy and computation integrity against colluding and malicious workers. Using Shamir's secret sharing, however, requires significant storage overhead, significant computation complexity and significant amount of randomness to provide data privacy compared to LCC (See Table I in [29]). This motivates us in this work to leverage verifiable computing in conjunction with LCC. Leveraging verifiable computing in machine learning has also been considered in [26]. In particular, Slalom [26] has been proposed recently which uses TEE-GPU collaboration for privacy-preserving inference. Because of the hardware limitations of TEE and consequently its lower performance, Slalom offloads the linear operations to an untrusted GPU and use Freivald's algorithm for verification [8] which is less computation intensive than the original computation in the TEE for verifying the linear computations of GPU. However, this scheme is designed for a single GPU system and cannot scale to the distributed system. Also, it does not support training and cannot address straggler mitigation.

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

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

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

In principle, VCC can be applied to the distributed computation of any polynomial f. However, VCC is particularly suitable for matrix-vector, matrix-matrix multiplications and in machine learning applications such as linear regression and logistic regression. The key idea of VCC is to use separate mechanisms for mitigating straggler effects and for tackling Byzantine nodes. VCC 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 $N - S$ to respond before starting to decode. Hence, VCC is highly parallelizable.

3 PROBLEM SETTING

In this section, we describe our system system, the threat models and our guarantees.

3.1 System Setting and Threat Model

We consider a distributed system with N nodes and a main server 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 the 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 the full control (root access) and design any attacks. 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.

3.2 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 straggles.
(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 and $\mathbf{X}_{\mathcal{T}}$ denotes the encoded datasets at the workers in $\mathcal{T}$.

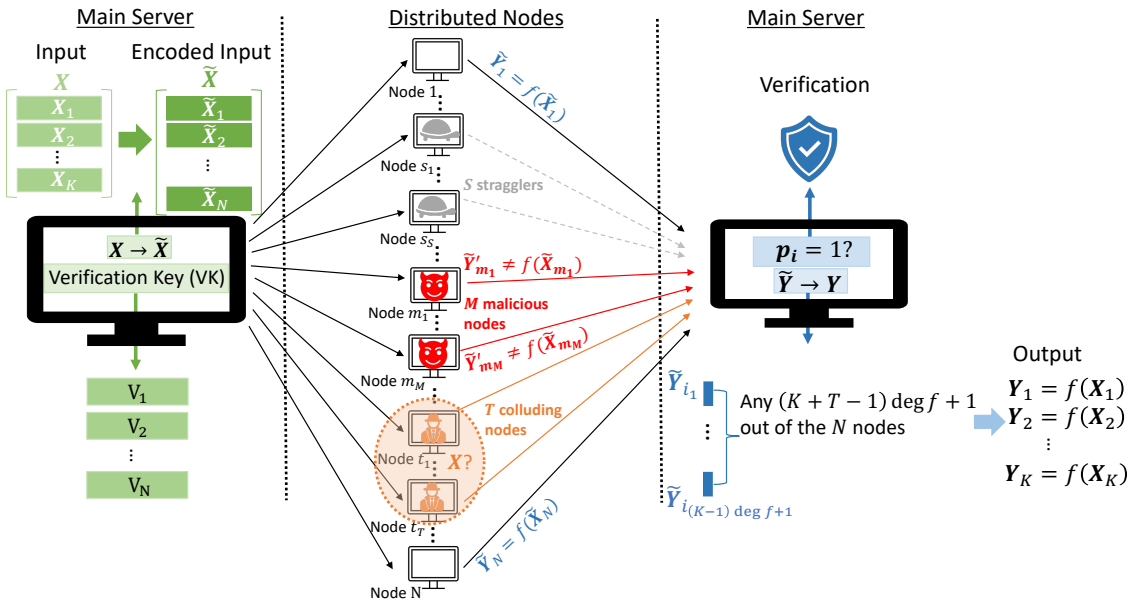

Fig. 2. Overview of the Verifiable Coded Computing (VCC) framework.

4 VERIFIABLE CODED COMPUTING

In this section, we present our verifiable coded computing 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 VCC.

4.1 Illustrating Example

We focus on the logistic regression problem to illustrate how VCC 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 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 function $h(\cdot)$ operates element-wise over the vector $\mathbf{X}\boldsymbol{w}^{(t)}$. In this example, we provide a two-round protocol as follows.

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 $\boldsymbol{e}^{(t)} = h(\boldsymbol{z}^{(t)}) - \boldsymbol{y}$. 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 master partitions the dataset $\mathbf{X}$ into K sub-matrices and encodes them using (N, K)-MDS coding. It then sends the coded sub-matrix $\tilde{\mathbf{X}}_i$ to the i-th worker, where $i \in [N]$.

(2) **Verification Key Generation.** The master server also computes a one-time verification key that helps in verifying the results of the computation results returned by the worker nodes afterwards. Specifically, the master 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 node $i \in [N]$.
The master then computes the following private verification keys as in the Freivalds' algorithm [8] 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 master 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 first round of iteration t, when the i-th node returns its result $\tilde{\boldsymbol{z}}_i^{(t)}$, which is $\tilde{\mathbf{X}}_i \boldsymbol{w}^{(t)}$ if this node is honest, the master 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 second round of iteration t, when the i-th node returns $\tilde{\boldsymbol{g}}_i^{(t)}$ which is $\tilde{\mathbf{X}}_i^\top \boldsymbol{e}^{(t)}$ if this node is honest, the master 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 $\tilde{\mathbf{z}}_i^{(t)}$ and $\tilde{\boldsymbol{g}}_i^{(t)}$ which requires $O(\frac{m}{K} d)$ arithmetic operations. Through this step, the master can identify the malicious nodes with high probability [18] as

$$\Pr[\boldsymbol{r}_i^{(1)} \cdot \tilde{\boldsymbol{z}}_i^{(t)} = \boldsymbol{r}_i^{(1)} \cdot \boldsymbol{z}_i'] \leq \frac{1}{q}, \tag{10}$$

$$\Pr[\boldsymbol{r}_i^{(2)} \cdot \tilde{\boldsymbol{g}}_i^{(t)} = \boldsymbol{r}_i^{(2)} \cdot \boldsymbol{g}_i'] \leq \frac{1}{q}, \tag{11}$$

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 master 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 nodes that fail the verification, they are effectively treated as stragglers and their results are not used for decoding.

(5) **Dynamic Coding.** Ignoring a Byzantine node's result comes at the cost of reduced straggler tolerance as the main server has to wait for additional verified results. In the original MDS coding strategy, $(N-K)$ stragglers can be tolerated. Hence, the main server at occasional intervals, may dynamically adjust the coding strategy for the next round of distributed computing. If original MDS straggler tolerance is still desired, the main server changes the coding strategy to $(N-1, K-1)$ to account for this Byzantine node. 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 tradeoff redundant work with Byzantine tolerance.

4.2 General Description

As shown in Subsection 4.1, VCC is particularly suitable for matrix-vector and matrix-matrix multiplications, because the information-theoretic verification schemes of such computations are efficient [18]. Such computations are essential in several machine learning applications, such as gradient descent, linear regression and logistic regression. However, in principle, VCC can be applied to any polynomial f. We now explain the encoding, the verification and the decoding of VCC.

(1) **Data Encoding.** In VCC, the dataset $\mathbf{X} = (\mathbf{X}_1^\top, \mathbf{X}_2^\top, \cdots, \mathbf{X}_K^\top)^\top$ is encoded as in LCC, but VCC 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 $\mathbf{W}_j$ is chosen uniformly at random. Such a polynomial can be constructed as follows

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

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

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

for $j \in [K+T]$. Next, another set of N distinct points denoted by $\mathcal{A} = \{\alpha_1, \cdots, \alpha_N\}$ is selected. 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

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

The main difference between the encoding in LCC and the encoding of VCC is that LCC requires that $N \geq (K+T-1)\deg f + S + 2M + 1$, whereas VCC 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 VCC, then we can encode the dataset using an (N, K) MDS code as illustrated in Subsection 4.1.
S-resiliency and T-privacy. Since VCC encodes the data in the same fashion as LCC, then VCC is S-resilient and T-private as LCC [29]. In the case where (N, K)-MDS coding is used for encoding, $S = N - K$ and $T = 0$.

(2) **Verification Keys Generation.** The master server also computes a random private verification key $\mathbf{V}_i$ for each worker node i which depends on $\tilde{\mathbf{X}}_i$ and the polynomial f.

(3) **Integrity Check**. When the master 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 master accepts the result with probability 1. Otherwise, the master detects this malicious behavior regardless of the computational power of this worker with high probability [18]. Specifically, we have

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

where the term $o(1)$ is a term that vanishes as the finite field size q grows.

(4) **Decoding.** Once the master collects $(K + T - 1) \deg f + 1$ verified results, then it 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 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 adopt 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 prepossessing 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

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

4.3 Applying VCC to Linear Regression using Lagrange Coding

To show how VCC can be generalized beyond linear computations, we focus on linear regression . 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 linear regression is to find the weight vector $\boldsymbol{w} \in \mathbb{R}^d$ that minimizes the cost function

$$C(\boldsymbol{w}) = \frac{1}{2m} \left\| \mathbf{X}\boldsymbol{w} - \boldsymbol{y} \right\|_2^2. \tag{20}$$

The gradient descent algorithm solves this problem iteratively by updating the model as

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \frac{\eta}{m} \mathbf{X}^{\mathrm{T}} (\mathbf{X}\boldsymbol{w}^{(t)} - \boldsymbol{y}), \tag{21}$$

where η is the learning rate. That is, linear regression corresponds to computing a degree-2 polynomial $f(\mathbf{X}) = \mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{w}$.

We now illustrate how VCC addresses this problem.

(1) **Data Encoding.** Before starting the computation, the main server encodes the dataset using Lagrange coding and sends the encoded sub-matrices to N workers.

(2) **Verification Keys Generation.** The main server also computes a one-time computation that helps in verifying the results of the nodes afterwards. Specifically, the master chooses a vector $\boldsymbol{r}_i \in \mathbb{F}_q^d$ uniformly at random for each node $i \in [N]$. The master then computes

$$\boldsymbol{s}_i \triangleq \boldsymbol{r}_i \tilde{\mathbf{X}}_i^{\mathrm{T}} \tilde{\mathbf{X}}_i \tag{22}$$

as a private verification key and keeps it along with $\boldsymbol{r}_i, \forall i \in [N]$.

(3) **Integrity Check.** At iteration t, when the i-th node returns its result, which is $\tilde{\boldsymbol{z}}_i^{(t)} \triangleq \tilde{\mathbf{X}}_i^{\mathrm{T}} \tilde{\mathbf{X}}_i \boldsymbol{w}^{(t)}$ for honest node, the master checks the equality

$$\boldsymbol{s}_i \cdot \boldsymbol{w}^{(t)} = \boldsymbol{r}_i \cdot \tilde{\boldsymbol{z}}_i^{(t)}. \tag{23}$$

This verification step is done in only $O(d)$ arithmetic operations and is much faster compared to computing $\tilde{\mathbf{z}}_i^{(t)}$ which requires $O(\frac{m}{K} d)$ arithmetic operations. Through this step, the master can identify the malicious nodes with high probability as

$$\Pr[\boldsymbol{r}_i \cdot \tilde{\boldsymbol{z}}_i^{(t)} = \boldsymbol{r}_i \cdot \boldsymbol{z}_i'] \leq \frac{1}{q}, \tag{24}$$

for any $\tilde{\boldsymbol{z}}_i' \neq \tilde{\boldsymbol{z}}_i^{(t)}$ [18].

(4) **Decoding.** After the main server verifies the results from at least $2K-1$ nodes, it can recover the final output.

5 EXPERIMENTS

We present an empirical study of the performance of VCC compared to the uncoded baseline. Our focus is on training a logistic regression model for image classification, while the computation load is distributed to multiple nodes on the DCOMP testbed platform that we gained access to [10]. In our experimental setup, we focus on the case where $T = 0$.

Experiment Setup. We train the logistic regression model described in Subsection 4.1 for binary image classification

on the GISETTE [12] dataset to experimentally examine two things: the the performance gain of VCC in terms of training time and accuracy, and the trade-off between various dynamic coding strategies. The size of the GISETTE dataset is $(m, d) = (6000, 5000)$ and the number of samples in the test set is 1000. Our experiments are deployed on a cluster of 13 Minnow instances on a DCOMP testbed, where 1 node serves as the master node 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 $(N, K) = (12, 9)$ configuration for the MDS coding in the experiment. The input matrix is divided into 9 sub-matrices, and then encoded into 12 partitions and assigned to the workers in the preprocessing stage. Each worker stores 1 of the 12 partitions. In the computation stage, each worker computes the product of its assigned matrix with the vector received from master, and then returns the result. The master then starts to verify any result upon receiving. Once the master collects 9 verified results, the master starts decoding the result.

The uncoded implementation is similar, except that 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 master waits for all 9 workers to return, and does not need to perform decoding to recover the result.

Algorithm 1 Pseudo-code for quantization

procedure QUANTIZATION($\mathbf{X}$)
    $\mathbf{X}_q$=Field(Round($\mathbf{X} \cdot 2^l$)) ▷ l is the quantization parameter

procedure ROUND($\mathbf{X}$)
    for $\forall X_i \in \mathbf{X}$ **do**
        if $X_i - \lfloor X_i \rfloor < 0.5$ **then**
            $X_i^r \leftarrow \lfloor X_i \rfloor$
        else
            $X_i^r \leftarrow \lfloor X_i \rfloor + 1$
    return $\mathbf{X}^r$

procedure FIELD($\mathbf{X}$)
    for $\forall X_i \in \mathbf{X}$ **do**
        if $X_i < 0$ **then**
            $X_i \leftarrow X_i + q$ ▷ two's complement to represent negative numbers in the finite field
        $X_i^f \leftarrow X_i \bmod q$
    return $\mathbf{X}^f$

Quantization and Parameter Selection. Since the MDS coding and the integrity check technique work over a finite field $\mathbb{F}_q$, but the inputs and weights for the model training are often defined in real domain, VCC needs to quantize the inputs and model weights to integers as $x^r = \text{round}(2^l \cdot x)$, 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 two's complement representation. The overall procedure of this quantization scheme is detailed in Algorithm 1. When the products are received by the master server, q is subtracted from all the elements larger than $\frac{q-1}{2}$ to restore negative numbers. The products are then scaled by 2^{-l} to convert them back to real numbers. There are prior works that use quantization schemes in training machine learning models [11, 21] without 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 size. The Minnow nodes use a 64-bit implementation, and the number of features in 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 $q = 2^{25} - 39$ to satisfy the limitation. Any model parameter that is larger than q will be quantized. In our experimental setup, the GISETTE dataset values are all non-negative integers and fit within the finite field $q = 2^{25} - 39$. Hence, no quantization was necessary. For the model weights, we then optimize the quantization parameter l to be 5. Note that the bias is implemented as part of the weights, so it shares the same quantization parameter as the model weights.

Byzantine attack models. We consider several Byzantine attack models that are widely used in previous works [16, 24].

- **Reversed Value Attack.** In this attack, Byzantine nodes that were supposed to send $\mathbf{z} \in F_q^{m/K}$ to the master instead send $-c\mathbf{z}$, for some $c > 0$. We set $c = 1$ in our experiment.
- **Constant Byzantine attack.** In this model, Byzantine nodes always send a constant vector to the master with dimension equal to that of the true product.

End-to-end Convergence Performance. We evaluate the end-to-end convergence performance of VCC, under two setups and two attack models, and compare it to the uncoded approach. In order to fulfill the requirement for $(12, 9)$-MDS coding that $12 \geq 9 + S + M$, we selected two setups to meet this restriction 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 1 node.
(2) $S = 2, M = 1$. In the second setup, we reduce the Byzantine tolerance to at most 1 node, while reducing straggler tolerance to 2 nodes.

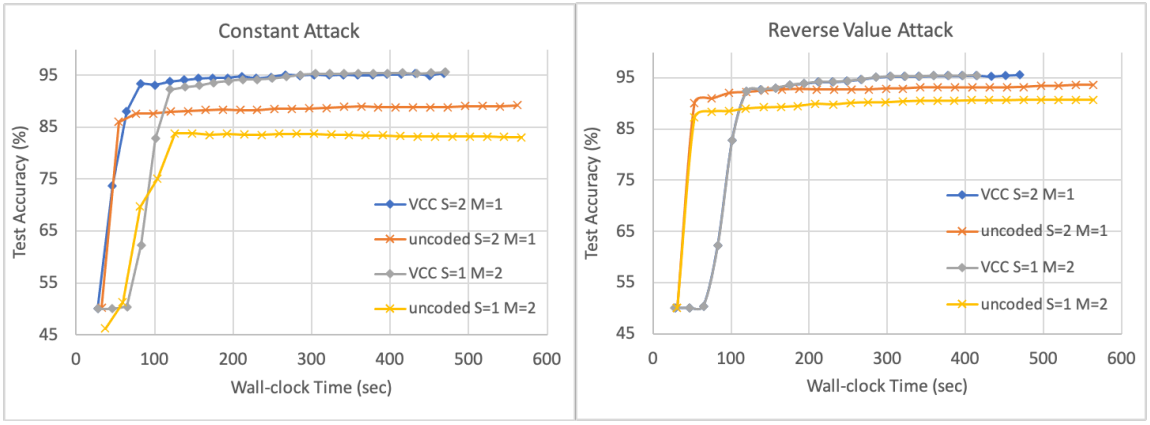

Fig. 3. End-to-end comparisons between VCC and uncoded baseline under (left) constant attack and (right) reverse value attack.

Both VCC and uncoded baseline are trained for 50 iterations. Fig. 3 shows how the test set accuracy varies with training time. As expected, the uncoded scheme may not converge to the same accuracy as VCC even if there is only one Byzantine node. In the constant attack model, the setup with $(S = 2, M = 1)$ converges at best to about 90% accuracy. The setup with $(S = 1, M = 2)$ converges to even a lower accuracy of 83%. This result is expected 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.

Under the constant attack model, applying VCC leads to significant improvement in Byzantine-robustness compared to the uncoded approach, and the test accuracy increases from 83% to 95.6%. Both the VCC setups eventually converge to high accuracy. In the $S = 1, M = 2$ setup, VCC takes longer to converge since it has to drop two of the Byzantine results consistently leading to more straggler exposure. The reason why the uncoded approach failed to converge to the same test accuracy under the constant attack model is that the master effectively lost information about a subset of the input assigned to the Byzantine nodes, and may not recover the desired optimal model.

We then consider the reverse value attack. Reverse value is a weaker attack, since the small values produced during the matrix-vector operations when added or subtracted do not derail the overall training convergence as dramatically. Hence, the uncoded approach achieves 93.6% test accuracy. VCC improves the test accuracy from 93.6% to 95.6%.

Applying VCC leads to significant end-to-end speedups. In particular, VCC achieves up to 6.8x speedup gain in the amount of time to achieve 89.2% test set prediction accuracy under constant attack, which is the highest accuracy achieved by baseline when $M = 2$ under constant attack. VCC achieves up to 4.7x speedup for achieving 90.7% test accuracy under reverse value attack.

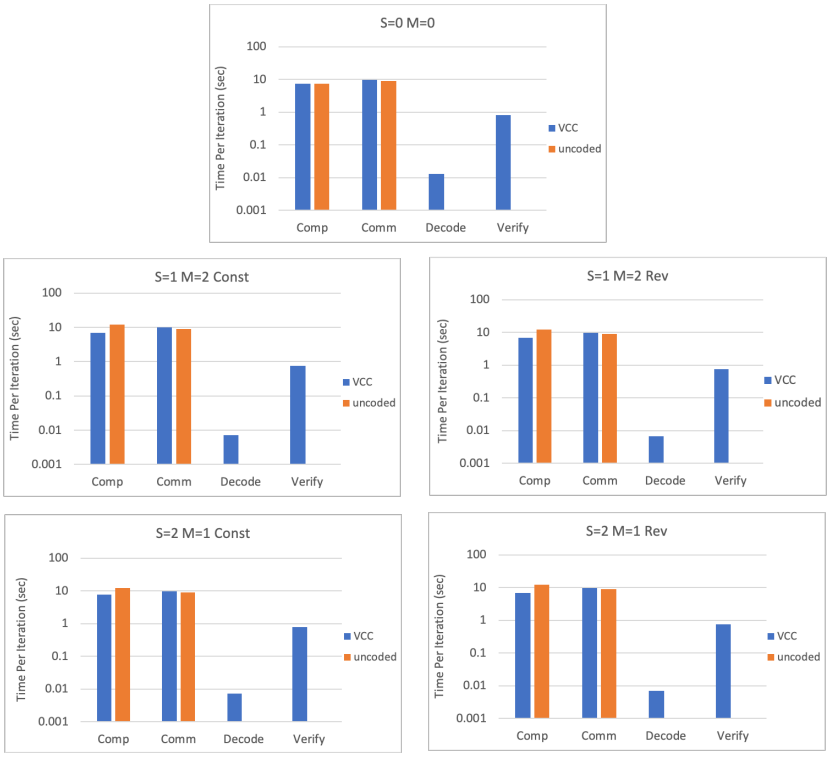

Fig. 4. Per iteration runtime analysis of VCC and uncoded baseline under different number of stragglers and malicious nodes.

Per Iteration Cost of VCC. The per iteration cost of applying VCC to logistic regression using GISETTE dataset is shown in Fig. 4. We breakdown the iteration cost into four categories: (1) compute time which is the worst case latency for performing the matrix operations across all the worker nodes, (2) communication time which includes the time to send and receive data between worker and the central server, (3) the decoding cost on the central server node (4) the

time to verify the results. In a straggler-free and Byzantine-free environment (the top-most chart), the decoding and verification time of VCC incurs extra latency. But when there are stragglers in the cluster, the decoding and verification overhead in VCC is dwarfed by the straggler latency. The overhead of VCC does not scale with the number of stragglers or Byzantine nodes, but the compute time of the uncoded approach is entirely left open to the vagaries of the straggler behaviour.

Dynamic Coding. We evaluate the performance of the two strategies in dynamic coding for various number of stragglers in our cluster as shown in Fig. 5. Each bar shows the average relative execution time spent by the application for 16 iterations, normalized by the execution time of the $(12, 9)$-MDS coded VCC scheme when there is no straggler in the cluster. The execution time composition is the same as described above. For the $(11, 9)$-MDS coded strategy, the execution time is approximately the same as the $(12, 9)$-MDS coded strategy and remains steady with one and two stragglers, but grows sub-linearly once the straggler count exceeds two, since it is designed to tolerate a maximum of two stragglers only and a significant penalty will be imposed when the redundancy is not enough. The $(11, 8)$-MDS code is a more conservative strategy than the $(11, 9)$-MDS code, as it has the same redundancy as the $(12, 9)$-MDS code. Although it increases the average execution time by around 4% as compared to the more aggressive $(11, 9)$ strategy when the cluster holds no more than two stragglers, it does not have to suffer from the extra 20% tail latency when there are more than two stragglers in the cluster.

Fig. 5. Dynamic coding execution time comparison

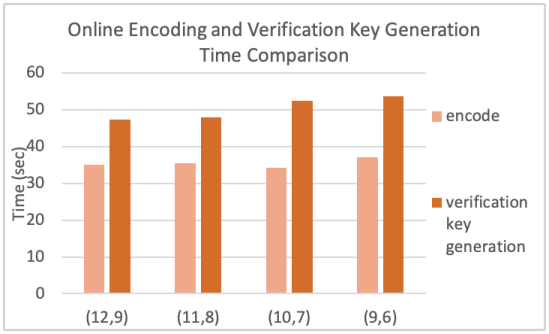

Fig. 6. Encoding and verification key generation time comparison

The re-encoding and verification key re-generation are expensive if performed online during the program execution, as shown in Fig. 6. However, such computation cost can be saved by generating the encoded data matrix and verification key offline before the application starts, and incurs certain storage overhead by storing them locally on the master server.

6 CONCLUSION

In this work, we presented VCC, a framework for resilient, robust, and private distributed training via coded computing and verifiable computing. VCC 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 uncoded distributed systems. Unlike prior coded computing approaches, VCC decouples the computational integrity check from the straggler tolerance thereby reducing the cost of Byzantine tolerance.

VCC 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, like Intel SGX 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.

REFERENCES

[1] 2021. Apache Hadoop. http://hadoop.apache.org/, last accessed on 05/01/2021.

[2] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective straggler mitigation: Attack of the clones. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 185–198.

[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 265–278.

[4] Elwyn R Berlekamp. 1966. *Non-binary BCH decoding*. Technical Report. North Carolina State University. Dept. of Statistics.

[5] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 118–128.

[6] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2018. Draco: Byzantine-resilient distributed training via redundant gradients. In *International Conference on Machine Learning*. PMLR, 903–912.

[7] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[8] R. Freivalds. 1977. Probabilistic Machines Can Use Less Running Time. In *IFIP Congress*.

[9] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. 2015. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review* 43, 1 (2015), 347–360.

[10] Ryan Goodfellow, Stephen Schwab, Erik Kline, Lincoln Thurlow, and Geoff Lawler. 2019. The DComp Testbed. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/cset19/presentation/goodfellow

[11] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[12] Isabelle Guyon, Steve Gunn, Asa Ben Hur, and Gideon Dror. 2004. Result Analysis of the NIPS 2003 Feature Selection Challenge. In *Proceedings of the 17th International Conference on Neural Information Processing Systems* (Vancouver, British Columbia, Canada) *(NIPS'04)*. MIT Press, Cambridge, MA, USA, 545–552.

[13] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. 2019. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 30–46.

[14] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2017. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory* 64, 3 (2017), 1514–1529.

[15] Hema Venkata Krishna Giri Narra, Zhifeng Lin, Ganesh Ananthanarayanan, Salman Avestimehr, and Murali Annavaram. 2020. 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, 453–463.

[16] Saurav Prakash and Amir Salman Avestimehr. 2020. Mitigating Byzantine Attacks in Federated Learning. *arXiv preprint arXiv:2010.07541* (2020).

[17] Shashank Rajput, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2019. DETOX: A redundancy-based framework for faster and more robust gradient aggregation. *arXiv preprint arXiv:1907.12205* (2019).

[18] Saeid Sahraei and A Salman Avestimehr. 2019. INTERPOL: Information theoretically verifiable polynomial evaluation. In *IEEE International Symposium on Information Theory (ISIT)*. 1112–1116.

[19] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. 2015. When do redundant requests reduce latency? *IEEE Transactions on Communications* 64, 2 (2015), 715–722.

[20] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[21] Jinhyun So, Basak Guler, and A Salman Avestimehr. 2020. Byzantine-Resilient Secure Federated Learning. *arXiv preprint arXiv:2007.11115* (2020).

[22] Jinhyun So, Basak Guler, and Salman Avestimehr. 2020. A Scalable Approach for Privacy-Preserving Collaborative Machine Learning. *Advances in Neural Information Processing Systems* 33 (2020).

[23] Mahdi Soleymani, Ramy E Ali, Hessam Mahdavifar, and A Salman Avestimehr. 2021. List-Decodable Coded Computing: Breaking the Adversarial Toleration Barrier. *arXiv preprint arXiv:2101.11653* (2021).

[24] Adarsh M Subramaniam, Anoosheh Heidarzadeh, and Krishna R Narayanan. 2019. Collaborative decoding of polynomial codes for distributed computation. In *2019 IEEE Information Theory Workshop (ITW)*. IEEE, 1–5.

[25] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 513–527.

[26] Florian Tramèr and D. Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. *ArXiv* abs/1806.03287 (2019).

[27] Xun Xu. 2012. From cloud computing to cloud manufacturing. *Robotics and computer-integrated manufacturing* 28, 1 (2012), 75–86.

[28] Chien-Sheng Yang and A Salman Avestimehr. 2021. Coded computing for secure Boolean computations. *IEEE Journal on Selected Areas in Information Theory* 2, 1 (2021), 326–337.

[29] Qian Yu, Songze Li, Netanel Raviv, Seyed Mohammadreza Mousavi Kalan, Mahdi Soltanolkotabi, and Salman A Avestimehr. 2019. Lagrange coded computing: Optimal design for resiliency, security, and privacy. In *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 1215–1225.

[30] Qian Yu, Mohammad Ali Maddah-Ali, and Salman Avestimehr. 2017. Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication. In *NIPS*.

[31] Liang Feng Zhang, Reihaneh Safavi-Naini, and Xiao Wei Liu. 2014. Verifiable local computation on distributed data. In *Proceedings of the 2nd international workshop on Security in cloud computing*. 3–10.