**GARFIELD: System Support for Byzantine Machine Learning**

El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis,* Sébastien Rouault

EPFL
firstname.lastname@epfl.ch

**Abstract**

Byzantine Machine Learning (ML) systems are nowadays vulnerable for they require trusted machines and/or a synchronous network. We present GARFIELD, a system that provably achieves Byzantine resilience in ML applications without assuming any trusted component nor any bound on communication or computation delays. GARFIELD leverages ML specificities to make progress despite consensus being impossible in such an asynchronous, Byzantine environment. Following the classical server/worker architecture, GARFIELD replicates the parameter server while relying on the statistical properties of stochastic gradient descent to keep the models on the correct servers close to each other. On the other hand, GARFIELD uses statistically-robust gradient aggregation rules (GARs) to achieve resilience against Byzantine workers. We integrate GARFIELD with two widely-used ML frameworks, TensorFlow and PyTorch, while achieving transparency: applications developed with either framework do not need to change their interfaces to be made Byzantine resilient. Our implementation supports full-stack computations on both CPUs and GPUs. We report on our evaluation of GARFIELD with different (a) baselines, (b) ML models (e.g., ResNet-50 and VGG), and (c) hardware infrastructures (CPUs and GPUs). Our evaluation highlights several interesting facts about the cost of Byzantine resilience. In particular, (a) Byzantine resilience, unlike crash resilience, induces an accuracy loss, and (b) the throughput overhead comes much more from communication (70%) than from aggregation.

**Keywords**— Distributed Machine Learning; Byzantine Fault Tolerance; Stochastic Gradient Descent.

*Corresponding author.
1 Introduction

Machine Learning (ML) is nowadays distributed [42, 46]. A major motivation is scalability. The quantity of data available to ML tasks is huge and can only be handled with distributed architectures. For instance, the size of Google’s ad impression log to train an ad click predictor could reach trillions of examples [37], each representing a high-dimensional feature vector. Such a dataset expands daily with new examples [44] in order to yield better models.

The modern approach to distribute an ML task is through the server/worker distinction [43]. A parameter server coordinates the distribution of the training task among a large set of worker nodes. The parameter server typically aggregates the workers’ gradients by merely averaging them [5, 12], following the standard workhorse optimization algorithm: Stochastic Gradient Descent (SGD) [55]. As the number of participating machines in the distributed setup increases, the probability of failure of any of these machines increases too. In distributed computing, the most general way to model such failures is to assume an adversary that can control a subset of the system and make it arbitrarily deviate from the normal execution: we talk about a Byzantine adversary and Byzantine failures [40]. This includes bogus software, faulty hardware, and malicious attacks. With the increasing use of ML in mission-critical applications [26, 11, 52], building robust systems against these kinds of failures becomes a necessity. For instance in a datacenter environment, processing units fail [22] and training data (uploaded by the users of a service) could be malicious. Using vanilla state machine replication (SMR) techniques to solve such a problem is impractical in the ML context [20].

Tolerating Byzantine workers without replicating them has been recently well studied in the convex (see e.g., [16, 8]) and non-convex (see e.g., [10, 51, 61, 66]) settings. A key idea is to replace the vulnerable averaging scheme, used by the parameter server to aggregate gradients coming from workers, by a statistically robust gradient aggregation rule (GAR), e.g., Median [62]. Several GARs were proposed, and they differ according to their computation cost, the assumptions they make on the dimension of the model, or the ratio of correct nodes [9, 65, 25].

However, these prior approaches have two major limitations. First, all of them so far considered the parameter server as a highly-available, trusted component. Consider a multi-branch organization, e.g., a bank or a hospital, while training an ML model among its branches, the misbehavior of the central server may stop the entire learning or even corrupt it (Section 5.5). Second, most Byzantine ML systems so far assumed bounds on computation and communication delays, i.e., synchrony [20, 51]. An adversary can delay messages exchanged by correct machines by overloading the network or trying to consume their CPU by sending them a lot of packets to process [18]. In this case, correct machines would be signaled crashing, impeding the learning process.

We present in this paper Garfield, a system that achieves Byzantine resilience for ML applications, in both convex and non-convex settings, without trusting any component in the network nor requiring its synchrony, addressing the limitations of existing Byzantine-resilient ML systems. Garfield follows the nowadays standard server/worker architecture [43] while replicating the server on multiple machines. Garfield leverages ML specificities to make progress despite consensus being impossible in such an asynchronous, Byzantine environment [29]. Assuming that correct workers compute an unbiased estimate of the correct gradient with a bounded vari-

\(^1\)Not to confuse with asynchronous training [21].
ance [13], GARFIELD uses statistically-robust gradient aggregation rules (GARs)\(^2\) to compute a gradient with bounded deviation from the real/correct gradient, filtering out the effect of Byzantine replies. Tolerating Byzantine servers on the other hand relies on replicating the parameter server on multiple machines in addition to introducing the contraction property, an original application of Median [62], to ensure models at correct parameter servers remain close to each other. Such a property allows for achieving Byzantine resilience without relying on consensus/total ordering, circumventing the impossibility result [29]. We formally prove that GARFIELD guarantees the safe convergence of the learning process, despite the presence of Byzantine machines [24].

The design of GARFIELD is modular and generic. Essentially, GARFIELD provides (a) tools to replicate the parameter server on multiple machines as well as (b) networking abstractions to handle communication between workers and servers. Concretely, GARFIELD offers abstractions to read (i.e., pull) gradients and models from other machines and write the updated values to its local machine. Furthermore, GARFIELD implements four state-of-the-art Byzantine-resilient aggregation rules, for both CPUs and GPUs, with consistent interfaces for using them. Such a variety of GARs offered by GARFIELD enables the practitioner to trade the number of Byzantine machines to tolerate with the convergence speed, given a model dimension, as we discuss in Section 3.3. We integrated GARFIELD with two widely-used ML frameworks: TensorFlow [5] and PyTorch [49], and this went through important implementation choices to promote the practicality of GARFIELD. For instance, we implemented specific schemes to obtain efficient and parallelizable versions of the Byzantine-resilient GARs, especially on GPUs. Moreover, we devised the notion of separate replicated graphs for TensorFlow rather than relying on its shared graph design, as the latter would be a killer in a fully Byzantine environment, i.e., without any trusted node [20]. Our implementation [4] provides transparency: applications can be augmented with Byzantine resilience without changing their interfaces.

We report on our evaluation of GARFIELD, addressing the question of the cost of Byzantine resilience in a distributed ML deployment when compared to a vanilla deployment where all components are trusted and assuming network synchrony. We consider various ML models, as well as different hardware, i.e., CPUs and GPUs. We also study the cost when considering different degrees of resilience. For example, we use GARFIELD components (e.g., networking abstractions and server replication tools) to build a crash-tolerant baseline, which tolerates crashes of parameter server replicas, without relying on SMR. We also compare GARFIELD against AggregaThor [20], a state-of-the-art system that achieves partial Byzantine resilience, namely, Byzantine workers’ resilience, assuming however a trusted parameter server.

Essentially, we show that Byzantine resilience introduces around 5% loss in the accuracy compared to non-Byzantine deployments. This is interesting because crash resilience does not introduce any such loss. Yet, such an accuracy loss is expected due to the statistical properties of the employed GARs. Such GARs ensure convergence only to a bounded region close to any of the local minima. Hence, a Byzantine-resilient deployment could not reach the exact same point reached by a non-Byzantine-resilient deployment. Such an observation is made obvious only with large-scale deployments, as we show later in Section 5. In terms of throughput, we quantify the overhead of Byzantine resilience to less than 3x compared to a vanilla deployment. We root this overhead mainly to communication. Concretely, our experiments show that commu-
cation accounts for around 70% of the overhead while the aggregation step contributes to 15% of such an overhead. We quantify the cost of adding Byzantine resilience to servers, compared to partial Byzantine resilience (i.e., tolerating only Byzantine workers) with a trusted server, to 43% and the cost of Byzantine resilience, compared to the crash–tolerant baseline, to 27%. In the latter case, the server is replicated but Byzantine behavior is not tolerated. Notably, using GPUs achieves a performance improvement of at least two orders of magnitude over CPUs.

Summary of contributions.
1. We introduce Garfield, a Byzantine-resilient ML system that does not assume any trusted component, nor any bound on communication or computation delays.
2. We integrate Garfield with TensorFlow and PyTorch, while achieving (a) transparency: ML applications need not change their interfaces to work with Garfield and (b) full-stack computations on CPUs and GPUs.
3. We evaluate our implementation of Garfield [4] against different baselines, analyzing the cost of Byzantine resilience in different scenarios.

2 Background

2.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) [55] is the most widely-used optimization algorithm in ML applications [17, 42, 5]. Many impressive results of the last decade, e.g., in image classification challenges [38, 58], used SGD optimize a neural network.

A neural network (NN) is a mere parameterized mathematical function, called model. A toy example: \( f(i) = \alpha i + \beta \), where the factors \((\alpha, \beta) \triangleq \mathbf{x} \in \mathbb{R}^d\) are called the parameters. To optimize an NN actually means to optimize a loss function \( L(\mathbf{x}) \in \mathbb{R} \), which measures “how incorrect the model, parameterized by \( \mathbf{x} \), is when labeling an input”.

SGD addresses the following optimization problem:

\[
\mathbf{x}_{opt} \triangleq \arg \min_{\mathbf{x} \in \mathbb{R}^d} L(\mathbf{x})
\]  

(1)

The procedure is iterative, at step \( k \):

1. Estimate the gradient \( G(\mathbf{x}^{(k)}, \xi) \), with a subset \( \xi \) of size \( b \) of the training set, called mini-batch. This is a stochastic estimation of the uncomputable, real gradient \( \nabla L(\mathbf{x}^{(k)}) \).

2. Update the parameters in the opposite direction of the estimated gradient. At the \( k^{th} \) iteration, called step:

\[
\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \gamma_k \cdot G(\mathbf{x}^{(k)}, \xi)
\]

(2)

The sequence \( \{\gamma_k \in \mathbb{R}_{>0}\} \) is called the learning rate.

2.2 Parameter Server Architecture

Estimating the gradient at \( \mathbf{x} \) is computationally expensive. It consists of averaging the \( b \) estimates of the gradient \( G(x, \xi_i) \), where \( \xi_i \) is the \( i^{th} \) pair (input, correct label)
from the mini–batch. Each \( G(x, \xi_i) \) involves a \textit{backpropagation} pass, so the amount of arithmetic operations to carry out to estimate \( G(x, \xi) \approx \nabla L(x^{(k)}) \) is \( \mathcal{O}(b \cdot d) \).

Fortunately, this gradient estimation is also easily parallelizable. The computation of \( G(x^{(k)}, \xi_i) \) could be done entirely in parallel by distributing the \( b \) computations \( G(x^{(k)}, \xi_i) \) over \( n \) machines. One widely used architecture enabling this distribution is the \textit{parameter server} scheme [43], illustrated in Figure 1, where a centralized server holds the parameters \( x \).

For each training step, the server first broadcasts the parameters to workers, which then share the heavy gradient estimation (i.e., each worker uses a mini–batch of size \( b/n \)). When a worker completes its estimation, it sends it to the parameter server. Finally, the server \textit{averages} the received estimations and updates the parameters \( x \), as in Equation 2.

2.3 Byzantine Resilience

In the parlance of classical distributed computing, a system tolerates a Byzantine fault when it copes with a machine that can deviate arbitrarily from the algorithm assigned to it [40]. Such a behavior abstracts any kind of failures, including software bugs, hardware defects, corrupted data [30], communication omissions, or even adversarial attacks. We consider the ML context where any of the machines contributing to the learning process, i.e., a worker or a server, can behave arbitrarily.

Previous work achieves Byzantine resilience through robust aggregation, e.g., [10, 41], redundant gradient computation along with coding schemes [15], combining both ideas [51], or performance–based ranking [63, 66]. \textsc{Garfield} follows the robust aggregation path, yet extends it with tolerating network asynchrony possibly misbehaving servers.

3 The Garfield Framework

We first introduce the problem we tackle and our assumptions, then we describe the \textit{gradient aggregation rules} (GARs) implemented by \textsc{Garfield}. Finally, we describe our protocol and discuss some intuitions about its Byzantine resilience.

3.1 Problem Definition

Contrary to the standard parameter server scheme [43] where the server is a single point of failure, \textsc{Garfield} assumes \( n_{w} \) workers and \( n_{ps} \) servers, among which up to
workers and $f_w$ servers can behave arbitrarily. The goal is for the correct nodes\(^3\) to learn despite the $f_w + f_p$ Byzantine nodes. More concretely, GARFIELD devises a learning procedure (Figure 2) that reaches a final parameter $x_{opt}$ close to the one that would have been reached without Byzantine workers. The problem is well-studied in Byzantine-free distributed systems, e.g., [33] and systems with Byzantine workers, e.g., [61, 10, 8, 63]. We list below the properties (using informal wording) that we identified to also tolerate Byzantine servers:

1. **Bounded deviation.** Parameter vectors on correct servers should not deviate too much from each other.

2. **Eventual contraction.** Parameter vectors on correct servers should eventually reach the same values.

3. **Advancement of learning.** Before convergence (i.e., before some step $t$), each correct parameter vector should come closer to the optimal solution after each step. In other words, the updating vector should have the same direction as the true gradient.

4. **Global confinement.** Correct servers’ parameter vectors should remain confined within a bounded region.

The first and the last properties constitute the safety requirements; the rest are the liveness requirements. All these requirements are both crucial and non-trivial to attain with the existing algorithms. For instance, one would think of a solution in which each parameter server works on its own by collecting gradients from all workers and applying a statistically-robust GAR to filter out Byzantine replies. Unfortunately, such a solution will not work as it will not ensure neither the bounded deviation nor the eventual contraction requirements and as a result, it is impossible to ensure advancement of learning in each step. Intuitively, if parameter vectors on correct servers become arbitrarily far from each other, then the gradients of one vector (i.e., model) cannot be used to update the state of another vector. As a result, this novel setup requires a crucial novel requirement, which is the agreement among correct servers on the parameter vector at each step. With the well-known impossibility result of agreement in Byzantine, asynchronous setup [29], we rely on the abstraction of approximate agreement [28] instead.

### 3.2 Assumptions

We assume that the training data is distributed identically and independently (i.i.d) on all workers [67]. We assume $n_{ps} \geq 3f_{ps} + 1$, where the relation between $n_{w}$ and $f_{w}$ depends on the employed GAR (see Section 3.3). We allow for a strong adversary that can coordinate Byzantine nodes and can delay communication or computation at correct nodes, but not indefinitely [14]. In this sense, we do not assume any upper bound on the communication nor the computation time, i.e., we assume an asynchronous network.

We assume the Byzantine nodes have strong, yet bounded, computation power and arbitrarily fast communication channels so as to compute the strong attacks presented in the ML literature [9, 25]. We also assume that these nodes have access to a subset of the training dataset and can overhear gradients sent by correct workers. We assume that honest nodes can authenticate the source of a message to prevent spoofing and

\[^3\text{When there is no ambiguity, we refer to both workers and parameter servers by nodes.}\]
Sybil attacks. This can be done using public-key signatures [53] and message authentication codes [59]. We assume that the adversary does not have enough computation power to break the cryptographic techniques [14].

3.3 Statistically Robust GARs

In this section, we introduce the GARs implemented by Garfield. A GAR is merely a function of \((\mathbb{R}^d)^q \rightarrow \mathbb{R}^d\), with \(d\) the dimension of the input vector (i.e., a gradient or a model) space \(\mathbb{R}^d\) and \(q\) the number of input vectors. Basically, these GARs wait for \(q\) vectors before applying some functions on them. Hence, in synchronous settings, these GARs can be deployed with \(q\) machines in the system (so that the aggregator node can gather replies from all nodes in the system within some time bound). Yet, in asynchronous settings, one would require to deploy \(q + f\) nodes to use these GARs, to ensure liveness of the protocol, where \(f\) denotes the maximum number of Byzantine inputs. In short, all GARs output a vector with special statistical properties that make them safe to use in the Byzantine setting.

1. **Median** [62] computes the coordinate-wise median among the input gradients and outputs one gradient of these medians. **Median** requires \(q \geq 2f + 1\), and its asymptotic complexity is \(O(qd)\).

2. **Multi–Krum** [20] assigns a score (based on a sum of distances with the closest neighbors) to each gradient a worker submits to the server, and then returns the average of the smallest scoring gradients set of size \(m\) (with \(m \leq q - f - 2\)). For aggregating workers’ replies, **Multi–Krum** achieves a better convergence rate compared to **Median** [20]. **Multi–Krum** requires \(q \geq 2f + 3\), and its asymptotic complexity is \(O(q^2d)\).

3. **MDA** [54] finds a subset group of gradients of size \(q - f\) with the minimum diameter among all other subsets, where the diameter of a group is defined as the maximum distance between any two gradients of this subset. **MDA** then outputs the average of the chosen subset. Notably, **MDA** carries an exponential\(^4\) asymptotic complexity, in \(O((q^f) + q^2d)\). Yet, as we will discuss later, its assumptions about variance are weaker than for the previous two GARs [47]. **MDA** requires \(q \geq 2f + 1\).

4. **Bulyan** [25] robustly aggregates \(q\) gradients by iterating several times (say \(k\) times) over another Byzantine–resilient GAR, e.g., **Multi–Krum**. In each of these \(k\) iterations, **Bulyan** extracts the gradients selected by such a GAR. Then, it computes the the coordinate-wise median of the \(k\) selected gradients. It then extracts the closest \(k'\) gradients to the computed median, and finally returns the coordinate-wise average of these \(k'\) gradients. Unlike previous GARs, **Bulyan** can sustain a model with a large dimension. Yet, it requires \(q \geq 4f + 3\), and its asymptotic complexity is \(O(q^2d)\).

**Tradeoffs.** In addition to the differences in the ratio of tolerated Byzantine nodes (inequalities relating \(q\) with \(f\)) and the computational cost of each GAR, the model dimension is also crucial in deciding which GAR to use. For high dimensions (e.g., order of millions), one should use **Bulyan** [25]. In low dimensions, the application

\(^4\)Exponential when \(f = O(q)\), polynomial when \(f = O(1)\).
Figure 2: One training loop in Garfield. The left line abstracts all parameter servers whilst the right line abstracts all workers. Arrows indicate communication. For instance, step 9 indicates exchanging the updated model among the server replicas. The GAR at step 7 could be any of those presented in Section 3.3.

setup should satisfy the variance assumption of the working GAR, as given below:

\[ \exists \kappa \in [1, +\infty], \forall (i, t, \theta) \in [1..n-f] \times \mathbb{N} \times \mathbb{R}^d, \]
\[ \kappa \Delta \sqrt{\mathbb{E} \left( \left\| g_i^{(t)} - \mathbb{E} g_i^{(t)} \right\|^2 \right)} \leq \| \nabla L(\theta) \|, \]

where,

\[ \Delta = \begin{cases} \frac{2f}{n-f} & \text{if GAR = MDA} \\ \sqrt{2} \left( n-f + \frac{f(n-f-2) + f^2(n-f-1)}{n^2 f^2} \right) \left( \frac{n-f}{n-f-2} \right) & \text{if GAR = Multi-Krum} \\ \sqrt{n-f} & \text{if GAR = Median}, \end{cases} \]

where \( g_i^{(t)} \) is the estimated gradient by worker \( i \) at time \( t \), and \( L(\theta) \) is the loss function at the model state \( \theta \). We provide a simple tool, `measure_variance.py`, to estimate whether such a condition is satisfied. Such a tool takes the experimental setup (\( n, f \), the batch size, etc) as inputs. Then, it does few training steps, while estimating the true gradient \( \nabla L(\theta) \) by computing a gradient using a huge batch size. In each training step, such a tool checks whether the condition stated above is satisfied or not, and then, gives the user statistics on how often such a condition is satisfied with each GAR.

### 3.4 The Garfield Protocol

Here we describe how Garfield works, highlighting how it tolerates Byzantine servers and workers while not assuming network synchrony.

Figure 2 displays a full training step and the communication involved.

**Initialization.** Each non-Byzantine parameter server initializes the model using the same seed so that each of them can start with the same initial model.
Worker behavior. Each worker (1) gathers a subset $q_{ps}$ of parameters from the first responding server replicas and (2) aggregates the received parameters using Median. Asynchrony, together with the presence of (at most) $f_{ps}$ Byzantine servers, makes it impossible for a worker to wait for more than $n_{ps} - f_{ps}$ parameter servers. The intuition is that if it does, then the (possibly colluding) Byzantine servers would only need to never reply to stop the training. Our protocol requires $2f_{ps} + 1 \leq q_{ps} \leq n_{ps} - f_{ps}$, and we consistently use $q_{ps} = 2f_{ps} + 3$ in our experiments.

Server behavior. Each server (1) gathers a subset $q_{w}$ of gradients from the first responding workers and (2) robustly aggregates these gradients using any Byzantine–resilient GAR (Section 3.3) with $q_{w} \leq n_{w} - f_{w}$ (due to asynchrony). The server then uses the aggregated gradient to update the parameters (Equation 2). Servers then perform one more round of communication between each other (steps 9–11 in Figure 2), in which they exchange their just–updated parameters. Each server waits for $q_{ps} \geq 2f_{ps} + 1$ other parameter vectors and then aggregates them with its own using Median. This additional step is paramount for correctness as it bounds the expected maximal pairwise distance between any two parameters, despite the impossibility of agreement in our asynchronous, Byzantine environment.

3.5 Correctness

The full formal specification of the Garfield protocol, along with its proof of correctness, is in [24]. There, we formally show why our generic protocol works when instantiated with each GAR. Here we sketch intuitions of the correctness.

Lemma 3.1. The expected maximum distance between parameter vectors on correct servers is always bounded.

Proof Sketch. Based on our assumptions, the correct workers compute an unbiased estimate of the real gradient with a bounded variance, following a correct execution of the SGD algorithm. Hence, the robust GAR (Step 7 in Figure 2) enables each parameter server to recover a correct gradient (within an expected bounded range) from $n_{w}$ workers when a minority is Byzantine. The key is that a robust GAR outputs a gradient which distance to any non-Byzantine gradient is bounded by a quantity independent from the received Byzantine gradient. For instance, when using MDA as a GAR, the following holds:

$$\forall l \in H, \|MDA(x_1 \ldots x_{n_w}) - x_l\| \leq \max_{(i,j) \in H^2} \|x_i - x_j\|,$$

where $x_k$ is the gradient estimated by worker $k$, and $l,i,$ and $j$ are indices of correct gradients belonging to the set $H$. Hence, the distance between any two aggregated gradients (on two correct servers) is bounded by the diameter of the correct gradients and so is the updated correct models (Step 8 in Figure 2), given that any correct server receives at least $f_{w} + 1$ correct gradients (as $q_w \geq 2f_{w} + 1$). Moreover, applying coordinate-wise Median\footnote{Note that other Median–based aggregation techniques could be also used like mean around median and trimmed mean.} with $q_{ps} \geq 2f_{ps} + 1$ ensures that the output model is bounded above and below by correct models, given at least $f_{ps} + 1$ correct models are gathered. This latter, fundamental piece for correctness is novel to the Byzantine SGD literature, which so far always relied on a single trusted server and only tolerated Byzantine fault at the workers’ side.\hfill \square
Lemma 3.2. **Garfield** ensures the learning advancement towards a local minimum in each step.

*Proof Sketch.* This relies on two fundamental pieces: (1) the computation of an unbiased gradient estimation on each correct worker and (2) the bounded deviation of parameter vectors on correct servers (Lemma 3.1). These two pieces result in the fact that the *effective gradients* (which are used at the end to update the parameter vectors) at the correct servers all lie in the same half-space as the true gradient. This ensures that all correct parameter vectors move towards a local minimum. Applying coordinate–wise *Median* (Step 10 in Figure 2) then outputs a vector in the range of correct vectors and hence, ensures that this output vector has advanced towards a local minimum. In [24], we formally prove the standard convergence criterion [13]:

\[
\lim_{t \to +\infty} E \| \nabla L (\bar{\theta}_t) \| = 0,
\]

(4)

where \(\bar{\theta}_t\) is the average of correct parameter vectors.

Lemma 3.3. Parameter vectors on correct servers are always confined beyond a bounded region.

*Proof Sketch.* By the construction of the algorithm, all correct parameter servers start with the same random vector. Based on Lemma 3.2, all vectors move in the same direction, which follows the same dynamics of vanilla SGD. Based on this, we can demonstrate that the second, third, and fourth moments of the *effective gradients* are bounded with respect to the norm of the parameter vector on each correct worker, following the same standard proof of SGD [13]. This essentially translates to the fact that the correct parameter vectors will remain confined within a bounded region.

Lemma 3.4. Eventually, all parameter vectors on correct servers contract almost–surely to the same values.

*Proof Sketch.* Based on Lemma 3.2, all correct parameter vectors move towards a local minimum. Given that the learning rate decreases with time, correct parameter vectors take smaller steps as they approach the local minimum. Moreover, based on Lemma 3.1, we know that correct models are always close to each other. As a result, after some time \(t\), the expected maximum distance between correct vectors decreases over time, contracting eventually to the same vector (as a result of the repetitive application of *Median* in each step). Formally, **Garfield** ensures that the correct model parameters almost–surely converge to the same value:

\[
\lim_{t \to +\infty} E \left( \max_{(a,b) \in [1 \ldots n_{ps} - f_{ps}]} \| \theta_t^{(a)} - \theta_t^{(b)} \| \right) = 0,
\]

(5)

where \(\theta_t^{(a)}\) is the parameter vector at server \(x\) at time \(t\).

Theorem 3.5. **Garfield** achieves Byzantine–resilient learning with malicious servers and workers.

*Proof Sketch.* Given all these previous lemmas, we can plug **Garfield** in the standard proof of SGD [13], showing convergence, despite the presence of Byzantine machines.
4 Garfield Design and Implementation

4.1 System Components

Garfield is designed in a modular way as shown in Figure 3. In this section, we describe each of these modules.

Controller. This module controls the training task. It is used for cluster deployment, definition of parameters, as well as for launching experiments. This encompasses parsing the cluster information, such as nodes' jobs (servers or workers) as well as their IPs and port numbers, starting the training procedure over SSH, and parsing experiments' parameters, e.g., the maximum number of Byzantine workers and servers.

Experiment. This module abstracts the available models and datasets for training. We implement a unified interface to the TensorFlow slim research models and TorchVision models. This enables us to experiment with various models, e.g., ResNet [32] and VGG [57]. On the training side, we leverage the compute and apply gradients functions to the underlying system, be it TensorFlow or PyTorch. We give unified interfaces for reading, writing, and updating the parameter vector. Having this fine-grained control on the training process is crucial in achieving Byzantine resilience.

Aggregation. We implement a highly-parallelized version of the four Byzantine-resilient GARs mentioned above, on both CPUs and GPUs, and we create wrappers (including dependency management, automatic compilation and loading) to use them as custom operations in both TensorFlow and PyTorch. Such wrappers make it possible to involve the GARs with the same interface for both frameworks, though the lower-level interfaces each framework provides differ substantially. To use a GAR, the common interface consists in two functions: init() and aggregate(). The init() function takes the name of the required GAR (e.g., “median”), the value of n, the total number of inputs, and f, the maximum number of Byzantine inputs. The second function, aggregate(), takes n tensors (could represent gradients or models) and outputs the aggregated one. Whether this function will execute on a CPU or a GPU depends on the device on which the input vectors are stored. In this way, our implementation abstracts the device, CPU or GPU, and the framework, TensorFlow or PyTorch, away from the developer, achieving modularity and our ease-of-use goal. We implement carefully optimized versions of the GARs. For example, aggregating gradients may require multiple iterations, calculating some distance-based scores for each of them in each iteration, e.g., with Multi-Krum or Bulyan. We then cache the results of each of these iterations (in the CPU or GPU memory) and hence, remove redundant computations. Besides, we reduce the memory cost by allocating space only for one iteration along with the intermediate selected gradients.

Network. Existing networking abstractions in both frameworks, TensorFlow and PyTorch, are not enough to be used (1) in a Byzantine, asynchronous environment and (2) with replicated parameter servers.\textsuperscript{6} For example, one can deploy distributed

\textsuperscript{6}Not to be confused with replicated graphs in TensorFlow. In our case, the replicated servers (all have the same graph) are independent, rather than shared between machines, and they do exactly the same computation on the same data for fault tolerance rather than for performance. Both kinds of replication can be combined, but this is out of the scope of this paper.

11
training on PyTorch by using `DistributedDataParallel()` or on TensorFlow by running `ParameterServerStrategy()`. However, both are high-level abstractions that cannot be tweaked to support both network asynchrony and parameter server replication.

Following the same model of the `Aggregation` module, we define the same interface in both frameworks to the networking layer. Basically, `Garfield` supports two functions: `get_gradients()` and `get_models()`. The `get_gradients()` function is used on the server side to read the updated gradients from the workers. It accepts two parameters: `t`, the index of the current iteration, and `qw`, the number of workers from which a server should receive replies with `qw ≤ nw`; `qw = nw` denotes synchronous communication with no faults in the system, i.e., a server is expecting to receive replies from all workers.

By default, the function returns the fastest `qw` replies it receives. The `get_models()` function works in the same way, yet with one difference: it accepts a third Boolean value that denotes whether to read the models from the workers or the servers. It returns the models of the fastest `q` replying machines. Such abstractions make it easy and natural to communicate with all machines in the network in both synchronous and asynchronous communication environments and both frameworks.

### 4.2 Implementation

First, we present how we implement the communication abstractions, i.e., `get_models()` and `get_gradients()` in TensorFlow and PyTorch respectively. Then, we show how we implement an efficient version of the `median` function (which is used in `Median` and `Bulyan` GARs) on GPUs.

**Communication in TensorFlow.** TensorFlow adopts the notion of dataflow graph in which all computations are defined in one graph, even if deployed in a distributed environment, where all participating nodes share this graph. This is a critical vulnerability in the Byzantine setting as Byzantine nodes can write and execute code on other nodes [20]. Also, this shared graph hides the data communication among workers and servers, reducing the programming flexibility and disallowing having more communication rounds, which is crucial for Byzantine resilience.

We follow another route in which all nodes create an independent, yet replicated graph. Though this design has high memory overhead, we believe it is necessary to tolerate adversarial behavior. In addition to resolving the vulnerability, this design allows for more flexible communication patterns among the participating nodes. To be compliant with the design of TensorFlow, we use gRPC for communication and

---

7 We also include GPU implementation of other GARs yet, we focus on `median` because its GPU implementation is challenging as we discuss.

8 We believe that such an overhead could be reduced if the environment is Byzantine-free.
protocol buffers [60] for serializing and deserializing data. We use the pull model for transferring data: when a node, be it a worker or a server, needs some data, it pulls this data from the other nodes. This is done by initiating a remote procedure call. Each node implements a server that serves these requests. We define the protocol buffers which encode data exchanged between participants. We parallelize the replicated communication between workers and servers for requesting gradients and updated models so as to reduce the communication time as much as possible. However, abandoning the highly optimized TensorFlow distributed runtime and using independent graphs on each node requires context switches between TensorFlow and Python runtimes (as protocol buffers cannot serialize tensors directly). Concretely, when a node is requested to send a gradient or a model, it serializes the requested data to a protocol buffer, exiting the TensorFlow graph/runtime. On the receiver side, a node deserializes the received bytes back to a Tensor. Experiments show that the overhead of these conversions (including memory copying) is non-negligible.

**Communication in PyTorch.** We implement the same abstractions in PyTorch, but with a slightly different design compared to the TensorFlow one. First, there is no context switch between PyTorch and Python since PyTorch gives communication abstractions that can be used directly on tensors. Second, we pipeline the communication with aggregation as PyTorch gives access to gradients of each layer in the deep network separately; this allows for better utilization of both network and computation devices and hence, better scalability. Third, our implementation automatically chooses the best communication backend between *nccl* and *gloo* to allow GPU-to-GPU communication when possible. This is a plus compared to TensorFlow as gRPC does not allow such a kind of communication. Finally, we use the notion of groups in distributed communication, and we choose groups carefully to enable the usage of the networking abstractions that allow GPU-to-GPU communication. For example, in the servers-to-workers communication step, we use *broadcast()* rather than *gather()* when using the *nccl* backend as *gather()* does not support GPU-to-GPU communication.9

**SIMT median function.** Our implementation of the *median* function on CPU is quite straightforward: each of the \( m \geq 1 \) available cores processes a continuous share of \( n/m \) coordinates. Then each core applies, for each coordinate of its share, *introselect* (or equivalent) by calling the standard C++ *std::nth_element*.

Nevertheless, even embarrassingly parallel algorithms like *median* would not necessarily benefit from running on GPGPUs (General–Purpose computing on Graphics Processing Unit). That is because modern GPGPUs, to achieve parallel execution on many threads while limiting instruction fetch costs, batch threads into groups of, e.g. 32 threads that execute the same instruction.10 Algorithms like *introselect* [48] are branch-intensive, with possibly many instructions executed in each branch, and so, fails to scale on GPUs.

Reminiscent of [36], our implementation of *median* is built around a primitive that orders 3 elements without branching, by the use of the *selection instruction*, which converts a predicate to an integer value. Let \( v \) be the table of size 3 to reorder by increasing values. Thanks to the selection instruction, we can compute \( c[3] = \{v[0] > v[1], v[0] > v[2], v[1] > v[2]\} \), where \( a > b \) is 1 if \( a > b \) else 0. Then the indices \( i[2] = \{ \)

---

9https://pytorch.org/docs/stable/distributed.html
10In case of branching, the threads execute in lock-step.
\[
\frac{(1+c[0]+2*c[1]+c[2]-c[1]\oplus c[2])}{2},
\frac{(4-c[0]-2*c[1]-c[2]+c[0]\oplus c[1])}{2}
\]
and finally the reordered values $w$ of $v$ is:
\[
w[3] = \{ v[i[0]], v[3-i[0]-i[1]], v[i[1]] \}.
\]
Using this reordering primitive, we manage to implement an efficient version of median with minimal branching.

5 Evaluation of Garfield

In the following, we first describe the settings we use and the baselines we compare Garfield against. Then, we show the convergence rate and throughput of Garfield in a Byzantine and a non-Byzantine environments. We defer more experiments to the supplementary materials.

5.1 Setting

Testbed. Our experimental platform is Grid5000 [2]. For the experiments deployed with CPUs, we employ nodes from the same cluster, each having 2 CPUs (Intel Xeon E5-2630 v4) with 10 cores, 256 GiB RAM and 2×10 Gbps Ethernet. For the GPU-based experiments, we employ nodes from two clusters (due to the limited number of nodes in one cluster); nodes in different clusters have different specifications. Each node has 2 identical GPUs.

Metrics. We evaluate the performance of Garfield using the following standard metrics:

Accuracy. This measures the top-1 cross-accuracy: the fraction of correct predictions among all the predictions, using the test set. Concretely, it shows the quality of the learned model over time. We measure the accuracy with respect to time and model updates.

Throughput. This quantifies the number of updates that the system processes per second, measured at the parameter server. For fault-tolerant systems that employ multiple parameter servers, we report the highest throughput, which corresponds to the fastest correct parameter server.

Application. We consider two variants of Garfield: the first uses Bulyan [25] to aggregate gradients, and hence achieves Byzantine resilience in high dimensions while assuming network asynchrony. The second one uses Multi-Krum [20] to aggregate gradients while assuming network synchrony. As the first instance provides stronger guarantees, it is more expensive to deploy as we show later. Unless otherwise stated, we use our TensorFlow version with the first application and PyTorch for the second one.

We consider the image classification task due to its wide adoption as a benchmark for distributed ML systems [17, 5]. We use MNIST [3] and CIFAR-10 [1] datasets. MNIST is a dataset of handwritten digits with 70,000 28 × 28 images in 10 classes. CIFAR-10 consists of 60,000 32 × 32 colour images in 10 classes. Table 1 presents the models we used for evaluation.
Table 1: Models used to evaluate Garfield.

| Model     | # parameters | Size (MB) |
|-----------|--------------|-----------|
| MNIST CNN | 79510        | 0.3       |
| CifarNet  | 1756426      | 6.7       |
| Inception | 5602874      | 21.4      |
| ResNet-50 | 23539850     | 89.8      |
| ResNet-200| 62697610     | 239.2     |
| VGG       | 128807306    | 491.4     |

Setup. For TensorFlow experiments, we employ 18 workers, out of them 3 could be Byzantine \((n_w = 18, f_w = 3)\) and 6 servers, 1 could be Byzantine \((n_{ps} = 6, f_{ps} = 1)\). We employ a batch size of 32 at each worker, leading to an effective batch size of 480 in the normal case \((n_w = 4f_w + 3 = 15)\). Recent studies show that going further than this number does not help achieve faster convergence [56], in addition to reducing throughput. For PyTorch experiment, we use 10 workers, with also 3 Byzantine, and 3 servers, with only 1 Byzantine. We use a batch size of 100 at each worker. We repeated all experiments multiple times and found that the error-bars are always very small compared to the presented values and hence, we omit them for better readability.

5.2 Baselines

To the best of our knowledge, Garfield is the first system to tolerate both Byzantine servers and workers. Hence, it is hard to find the perfect baseline to compare Garfield against. We chose the following baselines to show the benefits of deploying Garfield.

Vanilla baseline. This is the vanilla deployment of TensorFlow or PyTorch. We highlight two key differences between this deployment and Garfield. First, this setup uses only one trusted parameter server. This reduces the communication links with the workers and eliminates the overhead of synchronization required in case of using multiple servers. Second, a lightweight GAR i.e., Average is used by such a baseline, where more computational intensive ones e.g., Bulyan are used by Garfield. Thus, comparing Garfield against this baseline quantifies the overhead of replicating the parameter server and the Byzantine resilience cost.

AggregaThor [20]. This is the only existing scalable ML system that achieves Byzantine resilience, yet only for Byzantine workers. It is built on TensorFlow and supports training only on CPUs.\(^\text{11}\) AggregaThor uses only one trusted server while tolerating Byzantine workers. Moreover, it works only under a synchronous network assumption. For a fair comparison with our TensorFlow-based system, we run AggregaThor with Bulyan [25]. Thus, comparing with this baseline quantifies the overhead of the network asynchrony assumption and the replicated parameter server.

Crash-tolerant protocol. We implement a strawman approach to tolerate crash failures, assuming synchronous communication, using Garfield components. As worker crashes do not affect the learning convergence eventually, we only tolerate server crashes, by replicating the server. Server replicas get the updates from all

\(^{11}\)That is why it is missing in all GPU and PyTorch figures.
workers and average them each iteration, but workers contact only one of these replicas, i.e., the primary, to get the updated model. In the case of primary crash (signaled by a timeout), workers contact the next server, marking it as the new primary. The new primary sends its view of the model to all workers so as to inform them about the change. The model sent by the new primary could be outdated compared to the model of the crashed primary (due to missing some updates). This is still fine and the learning will converge eventually [64], given that \( n_{ps} \geq f_{ps} + 1 \) where \( n_{ps} \) is the total number of replicas and \( f_{ps} \) is the maximum number of crashing nodes, i.e., servers. Thus, this deployment guarantees eventual convergence, without any guarantees on throughput nor convergence rate. Some ML systems already use Paxos [39] for crash fault tolerance [17, 43]. However, this algorithm, we believe, gives strictly weaker guarantees (in terms of consistency of model state among replicas), and hence has a better performance than Paxos.\(^{12}\) Comparing with this baseline quantifies the cost of Byzantine resilience.

\(^{12}\)Unfortunately, the code is not open-sourced for comparison.
respect to the number of inputs/gradients (i.e. \( n \) the number of workers or servers) and the gradient dimension (i.e. \( d \)).

Vanilla frameworks, e.g. TensorFlow, average gradients at the parameter server\(^{13}\) and perform no aggregation of parameter vectors, as there is only one parameter server. So we also include the evaluation of Average, which has been implemented as a part of the GARFIELD library. We use Average as a baseline to our GARs implementation.

For a fair comparison, we set \( f \), the number of declared Byzantine inputs, to \( \left\lfloor \frac{n-3}{4} \right\rfloor \) for all Byzantine-resilient GARs and hence, the smallest possible \( n \) is 7. We set \( d = 10^7 \) in Figure 4a and \( n = 17 \) in Figure 4b. The metric for this micro-benchmark is the aggregation time: it includes the aggregation of \( n \) input vectors (all resident in GPU memory) and the transfer of the resulting vector back to main memory. Each point is the average of 21 runs, for which we observed a standard deviation two orders of magnitude below the observed average. We ran this micro-benchmark on an Intel Core i7-8700K CPU and two Nvidia GeForce 1080 Ti GPUs.

Theoretically, the asymptotic complexities of MDA, Multi-Krum, Bulyan, and Average are respectively \( \mathcal{O}\left(\binom{n}{f} + n^2d\right) \) [25], \( \mathcal{O}(n^2d) \) [10], \( \mathcal{O}(n^2d) \) [25] and \( \mathcal{O}(nd) \). Our implementation of Median has a best case complexity of \( \mathcal{O}(nd) \) and worst case of \( \mathcal{O}(n^2d) \). In practice for a fixed \( d \) (Figure 4a), we observe these asymptotic behaviors for Multi-Krum and Bulyan: quadratic in \( n \). Median shows good scalability with \( n \), maintaining a consistent performance that is very close to Average. Although the asymptotic complexity of MDA is exponential, our implementation achieves only a quadratic growth with \( n \). The values of \( n \) and \( f \) used in these experiments are merely too low to expose such a behavior, i.e., the exponential growth with \( n \). Average aggregation time remains roughly constant for a fixed \( d \) and \( n < 15 \), with an aggregation time of \( \sim 8 \) ms, and then grows linearly. For a fixed \( n \) (Figure 4b), we observe a linear time increases with respect to \( d \) for every one of the studied GARs.

5.4 Convergence Comparison

Figure 5 shows the results of two experiments: The first one (Figure 5a) trains CifarNet on TensorFlow-based systems (including AggregaThor and GARFIELD-based deployments) using CPUs, where the second one (Figure 5b) trains ResNet50 on PyTorch-based systems using GPUs; both experiments use CIFAR-10 as a dataset. The first experiment puts GARFIELD in a head-to-head comparison with the state-of-the-art Byzantine ML system, i.e., AggregaThor. The second experiment is an instance of a deployment of GARFIELD while training a bigger model using GPUs. Figure 5a shows that all systems achieve almost the same final accuracy. The Byzantine-resilient protocols converge slower than those using averaging during training. Yet, they manage to achieve the same accuracy as the vanilla deployment eventually i.e., after doing enough number of iterations. Figure 5a also shows that deploying GARFIELD does not add much overhead compared to the crash-tolerant one, in terms of the number of iterations till convergence (less than 1%). Surprisingly, GARFIELD achieves better final accuracy than AggregaThor. Figure 5b shows that GARFIELD fails to reach the same final accuracy as vanilla PyTorch (or even the crash-tolerant algorithm that uses averaging for aggregating gradients). This accuracy loss, although not clear in Figure 5a, makes sense as a direct byproduct of using a Byzantine-resilient GAR to aggregate workers’ gradients. Such a result highlights the accuracy loss introduced by

\(^{13}\text{www.tensorflow.org/api_docs/python/tf/train/XmlReplicasOptimizer} \)
5.5 Byzantine Behavior

As a check to our implementation, we conduct experiments with real Byzantine behavior, where we apply attacks on the vanilla baseline (PyTorch in this experiment), Garfield, and the crash-tolerant protocol. Figure 6 shows two kinds of attacks on both servers and workers sides. In the first attack (Figure 6a), the Byzantine node, be it a worker or a server, replaces its data, be it a gradient or a model, with random values, while in the second attack (Figure 6b), such vectors are reversed and amplified (multiplied by -100). We train CifarNet with 11 workers and 3 servers (in case of fault–tolerant algorithms) with 1 Byzantine node from each party. We do the training only for 20 epochs, rather than till convergence. On the one hand, both the vanilla deployment and the crash–tolerant deployment fail to learn under both attacks. On the other hand, Garfield manages to train the model safely and converges to a normal, high accuracy.

5.6 Throughput

We show here the computation and communication costs of Byzantine resilience. First, we quantify the overhead of employing Garfield compared to other baselines by measuring the throughput while training several models. Then, we analyze the scalability of Garfield with a different number of workers, Byzantine workers, and Byzantine servers. In this section and without loss of generality, we use ResNet-50 as our model. We do not employ any attack nor Byzantine behavior in these experiments as we want to quantify the overhead of Garfield in a normal, optimistic environment. Thus, we denote here the number of declared Byzantine nodes with the number of Byzantine nodes (or $f_w$ and $f_ps$).

Model dimension. Figure 7 depicts the cost of Byzantine resilience in terms of throughput. The throughput of the fault-tolerant systems is normalized to the vanilla baseline throughput in each case. Thus, the y-axis represents the slowdown that each of the fault-tolerant systems induces compared to the vanilla baseline deployment. 

\[\text{Figure 6: Garfield tolerance to two Byzantine attacks.}\]

Byzantine–resilient algorithms while training big models, as such a loss is not clear with training the small model.

\[\text{Byzantine Behavior}\]

More sophisticated attacks were proposed recently to trick several GARs we are using in Garfield under specific conditions and assumptions [9, 65]. Such attacks could be tolerated by hyper-parameter tuning [47]. We note that our protocol is based on the resilience guarantees provided by such GARs; resisting these attacks are out of the scope of this paper.
Figure 7: Slowdown of fault-tolerant systems normalized to the vanilla baseline (i.e., TensorFlow) throughput.

The cost of crash tolerance ranges from 1.4% to 161%, that of AggregaThor ranges from 1% to 141%, and that of GARFIELD ranges from 43% to 187% compared to the vanilla deployment. More interestingly, the GARFIELD overhead ranges from 6% to 27% compared to the crash-tolerant deployment and from 5% to 43% compared to AggregaThor.

Based on both figures, we extract several observations. First, the cost of GARFIELD compared to the vanilla baseline seems big (∼ 3x) however, the cost of Byzantine resilience is reasonable compared to weaker alternatives: interestingly, the cost of workers’ strong Byzantine resilience (using AggregaThor) is less than that of crash tolerance in most of the cases (more clear with big models). This leads to our next observation: ML training, especially on GPUs, is network-bound rather than compute-bound. This is confirmed in previous work [69, 34, 31]. Thus, noting the aggregation rules do not affect much the throughput (they always account for less than 15% of the total iteration time; see Figure 8), the systems requiring less communication would naturally achieve higher throughput.
Overhead breakdown. We pick one instance and take a closer look at the Garfield deployment to understand the main factors affecting its performance. Concretely, we run the same experiments while training ResNet-50, breaking the average latency per iteration, to understand the factors driving the overhead.

Figure 8 depicts the breakdown of the systems overhead when deployed on the CPU-based cluster. It is hard to decompose communication and computation time for TensorFlow. It is also not directly supported by AggregaThor to report the decomposition between communication, computation, and aggregation. Thus, the blue-and-orange bar in both is the time spent in both communication and computation combined. We infer the aggregation time of AggregaThor through other experiments; the green bar of AggregaThor is an estimate for the aggregation time.

Although abstractly all systems do the same computation, the time spent in Computation (the blue bar) is different in all of them. Practically, all of the fault-tolerant systems build layers on top of TensorFlow. The difference in computation times is spent in work like switching context between TensorFlow and Python (which may include copying some data) for instance. We quantify this additional computation overhead, in case of Garfield, to 36% compared to the vanilla TensorFlow deployment.

We did the same experiment with our PyTorch-based implementation using GPUs, and we got similar results (see Section 4 in the supplementary materials).

Number of workers. Increasing the number of workers ($n_w$), and hence increasing the effective batch size, is crucial in scaling distributed ML applications. Figure 9 depicts the scalability of Garfield with respect to the vanilla deployments. In this figure, throughput is measured in batches/sec rather than updates/sec since employing more workers allows increasing the number of batches processed per iteration. We draw three main observations from this figure. First, all systems scale with employing more workers, with around three orders of magnitude better throughput with GPUs.
compared to CPUs. Second, the throughput gap between the vanilla deployments and the fault-tolerant deployments increases with increasing \( n_w \), keeping the slowdown introduced by GARFIELD almost constant. Third, scalability of GARFIELD is almost as good as that of the crash-tolerant deployment and the difference in throughput with increasing \( n_w \) is almost constant. This shows that Byzantine resilience does not harm scalability compared to crash resilience.

![Figure 10: Throughput with increasing \( f_w \) and \( f_{ps} \).](image)

**Number of Byzantine workers.** As increasing the number of Byzantine workers \( (f_w) \) does not call for increasing the total number of workers, we fix \( n_w \) and hence, fixing the effective batch size in all cases. Increasing \( f_w \) leads to increasing the size of quorum which the servers wait for before proceeding to the aggregation phase. Thus, a higher value of \( f_w \) slightly decreases the throughput in the case of stragglers; this is confirmed in Figure 10a with \( f_w = 3 \).

**Number of Byzantine servers.** Increasing the number of Byzantine servers \( (f_{ps}) \) calls for increasing the total number of the server’s replicas \( (n_{ps}) \) so as to follow the GARFIELD condition: \( n_{ps} \geq 3f_{ps} + 1 \) (to account for asynchrony). Increasing \( f_{ps} \) introduces new communication links, which leads to throughput drop as shown in Figure 10b. This throughput drop is confirmed in the state machine replication (SMR) literature [35, 6], and we believe that the amount of drop (less than 45% in our case) is reasonable compared to what is reported before in the literature of SMR [19]. The assumption of 1 faulty parameter server introduces an overhead of 33% to achieve Byzantine resilience (compared to the crash-tolerant variant), which is reasonable. Finally we note that increasing \( f_{ps} \) does not affect the number of iterations required for convergence according to the GARFIELD protocol.

### 6 Related Work

With the impracticality (and sometimes impossibility [29]) of applying exact consensus to ML applications, approximate consensus [28] seems to be a good candidate. In approximate consensus, all nodes try to decide values that are arbitrarily close to each other and that are within the range of values proposed by correct nodes. Several approximate consensus algorithms were proposed with different convergence rates, communication/computation costs, and tolerable number of Byzantine nodes, e.g., [23, 27, 7, 45].

Inspired by approximate consensus, several Byzantine-resilient ML algorithms have been proposed to resist attacks in the synchronous non-convex setting. All try to mathematically bound the deviation of the aggregated gradient from the correct ones.
while assuming a trusted parameter server. Krum [10] employs a median-like aggregation rule. Multi-Krum [20] generalizes the idea by averaging more gradients to benefit from additional workers. Bulyan [25] addresses an attack that can fool some Byzantine-resilient algorithms by having them converge to a stable, yet faulty, state. [68, 62] consider different variants of robust mean-based algorithms under different assumptions and scenarios. Kardam [21] uses filtering mechanisms to achieve Byzantine resilience in an asynchronous training setup. Zeno [63] and Zeno++ [66] achieve Byzantine resilience using a performance–based ranking approach for the workers’ gradients in synchronous and asynchronous settings respectively. Draco [15], uses a coding scheme to restore correct gradients using redundant computations. Detox [51] extends this idea by combining coding schemes with robust aggregation to hit the sweet spot in the resilience–optimality spectrum. AggregaThor [20] resembles GARFIELD in that it relies on robust aggregation to achieve resilience. GARFIELD extends these efforts by tolerating Byzantine servers and workers under asynchronous communication. The problem of tolerating benign (e.g., crash) failures of parameter vectors was also addressed in the literature. Qiao et al. [50] leverages the self–correcting behavior of SGD to tolerate such failures. Other proposals addressed the problem from a systems perspective by making the parameter server crash-resilient [43, 17] using Paxos [39]. Others relied on checkpoints or live replication [5] of the parameter server. However, using such tools fail in the Byzantine context [29].

7 Conclusion

This paper presented GARFIELD, a system that provably achieves Byzantine resilience for Machine Learning (ML) applications without trusting any component nor requiring network synchrony. At the heart of GARFIELD lies a novel protocol that leverages statistically-robust gradient aggregation rules (GARs) to achieve Byzantine resilience to both servers and workers while replicating the parameter server on multiple machines. GARFIELD has a modular and a generic design, where it allows for a full-stack computation on both CPUs and GPUs. We integrated GARFIELD with TensorFlow and PyTorch while ensuring application transparency: the Byzantine resilience of ML applications can be provided without changing their original interfaces. Our evaluation of GARFIELD showed that Byzantine resilience, unlike crash resilience, induces an inherent loss in the final accuracy and that the throughput overhead of Byzantine resilience is moderate compared to crash resilience. Furthermore, we showed that GARFIELD’s overhead is moderate compared to a state–of–the–art system, AggregaThor, that achieves partial Byzantine resilience.

Acknowledgment

This work has been supported in part by the Swiss National Science Foundation (FNS grant 200021_182542/1).

References

[1] Cifar dataset. https://www.cs.toronto.edu/~kriz/cifar.html.
[2] Grid5000. https://www.grid5000.fr/.
[3] Mnist dataset. http://yann.lecun.com/exdb/mnist/.

[4] Garfield source code. https://github.com/LPD-EPFL/Garfield.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In OSDI, 2016.

[6] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. ACM SIGOPS Operating Systems Review, 39:59–74, 2005.

[7] Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In International Conference On Principles Of Distributed Systems, pages 229–239. Springer, 2004.

[8] Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. Byzantine stochastic gradient descent. In Neural Information Processing Systems, to appear, 2018.

[9] Moran Baruch, Gilad Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning. arXiv preprint arXiv:1902.06156, 2019.

[10] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In Neural Information Processing Systems, pages 118–128, 2017.

[11] Cara Bloom, Joshua Tan, Javed Ramjohn, and Lujo Bauer. Self-driving cars and data collection: Privacy perceptions of networked autonomous vehicles. In Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017), pages 357–375, 2017.

[12] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 1175–1191. ACM, 2017.

[13] Léon Bottou. Online learning and stochastic approximations. Online learning in neural networks, 17(9):142, 1998.

[14] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In OSDI, volume 99, pages 173–186, 1999.

[15] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. Draco: Byzantine-resilient distributed training via redundant gradients. In International Conference on Machine Learning, pages 902–911, 2018.

[16] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. arXiv preprint arXiv:1705.05491, 2017.

[17] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In OSDI, volume 14, pages 571–582, 2014.

[18] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In NSDI, volume 9, pages 153–168, 2009.
[19] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 177–190. USENIX Association, 2006.

[20] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Guirguis, and Sébastien Rouault. Aggregather: Byzantine machine learning via robust gradient aggregation. In SysML, 2019.

[21] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki, et al. Asynchronous byzantine machine learning (the case of sgd). In ICML, pages 1153–1162, 2018.

[22] Jeff Dean. Designs, lessons and advice from building large distributed systems.

[23] Danny Dolev, Nancy A Lynch, Shlomit S Pinter, Eugene W Stark, and William E Weihl. Reaching approximate agreement in the presence of faults. Journal of the ACM (JACM), 33(3):499–516, 1986.

[24] El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, Lê Nguyên Hoang, and Sébastien Rouault. Genuinely distributed byzantine machine learning. In Proceedings of the 39th Symposium on Principles of Distributed Computing, pages 355–364, 2020.

[25] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. The hidden vulnerability of distributed learning in Byzantium. In Jennifer Dy and Andreas Krause, editors, Proceedings of the 35th International Conference on Machine Learning, volume 80 of Proceedings of Machine Learning Research, pages 3521–3530, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[26] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. Nature, 542(7639):115, 2017.

[27] AD Fekete. Asymptotically optimal algorithms for approximate agreement. In Proceedings of the fifth annual ACM symposium on Principles of distributed computing, pages 73–87, 1986.

[28] Alan David Fekete. Asynchronous approximate agreement. In Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, pages 64–76, 1987.

[29] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. JACM, 32(2):374–382, 1985.

[30] Amirmasoud Ghiassi, Taraneh Younesian, Zhihong Zhao, Robert Birke, Valerio Schiavoni, and Lydia Y Chen. Robust (deep) learning framework against dirty labels and beyond. In 2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), pages 236–244. IEEE, 2019.

[31] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. arXiv preprint arXiv:1803.03288, 2018.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. CoRR, abs/1603.05027, 2016.
[33] Lie He, An Bian, and Martin Jaggi. Cola: Decentralized linear learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, Advances in Neural Information Processing Systems 31, pages 4536–4546. Curran Associates, Inc., 2018.

[34] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Komnios, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In NSDI, pages 629–647, 2017.

[35] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In USENIX annual technical conference, volume 8. Boston, MA, USA, 2010.

[36] M. Kachelrieß. Branchless vectorized median filtering. In 2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC), pages 4099–4105, Oct 2009.

[37] Larry Kim. How many ads does google serve in a day? URL http://goo.gl/oIidXO. http://goo.gl/oIidXO, 1(1), 2012.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Neural Information Processing Systems, pages 1097–1105, 2012.

[39] Leslie Lamport et al. Paxos made simple. ACM Sigact News, 32(4):18–25, 2001.

[40] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. TOPLAS, 4(3):382–401, 1982.

[41] Liping Li, Wei Xu, Tianyi Chen, Georgios B Giannakis, and Qing Ling. Rsa: Byzantine-robust stochastic aggregation methods for distributed learning from heterogeneous datasets. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pages 1544–1551, 2019.

[42] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In OSDI, volume 1, page 3, 2014.

[43] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In Big Learning NIPS Workshop, volume 6, page 2, 2013.

[44] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. Ad click prediction: a view from the trenches. In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1222–1230. ACM, 2013.

[45] Hammurabi Mendes, Maurice Herlihy, Nitin H. Vaidya, and Vijay K. Garg. Multidimensional agreement in byzantine systems. Distributed Computing, 28(6):423–441, 2015.

[46] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. JMLR, 17(1):1235–1241, 2016.

[47] El Mahdi El Mhamdi, Rachid Guerraoui, and Arsany Guirguis. Fast machine learning with byzantine workers and servers. arXiv preprint arXiv:1911.07537, 2019.
[48] David R Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[50] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault tolerance in iterative-convergent machine learning. In *International Conference on Machine Learning*, pages 5220–5230, 2019.

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

[52] Qing Rao and Jelena Frtunikj. Deep learning for self-driving cars: chances and challenges. In 2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS), pages 35–38. IEEE, 2018.

[53] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[54] Peter J Rousseeuw. Multivariate estimation with high breakdown point. *Mathematical statistics and applications*, 8:283–297, 1985.

[55] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[56] Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

[57] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[58] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[59] Gene Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.

[60] Kenton Varda. Protocol buffers. *https://github.com/protocolbuffers/protobuf*.

[61] Pooja Vyavahare, Lili Su, and Nitin H Vaidya. Distributed learning with adversarial agents under relaxed network condition. *arXiv preprint arXiv:1901.01943*, 2019.

[62] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Generalized Byzantine-tolerant sgd. *arXiv preprint arXiv:1802.10116*, 2018.

[63] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Zeno: Byzantine-suspicious stochastic gradient descent. *arXiv preprint arXiv:1805.10032*, 2018.

[64] Cong Xie, Oluwasanmi O Koyejo, and Indranil Gupta. Faster distributed synchronous sgd with weak synchronization. 2018.
[65] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Fall of empires: Breaking byzantine-tolerant sgd by inner product manipulation. *arXiv preprint arXiv:1903.03936*, 2019.

[66] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Zeno++: Robust fully asynchronous sgd. *arXiv preprint arXiv:1903.07020*, 2019.

[67] Eric P Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *KDD*, pages 1335–1344, 2015.

[68] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. *arXiv preprint arXiv:1803.01498*, 2018.

[69] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX ATC*, pages 181–193, 2017.
Additional Experiments

1 Convergence with Time

As discussed in the main manuscript, the end metric that one would use to assess the training progress is convergence over time. Such a metric combines both metrics presented in the main manuscript: convergence per epoch and throughput. Figure 11 is the mirror of Figure 4 in the main manuscript, which shows the convergence of Garfield compared to the baselines with the training rounds. Figure 11 shows the results of the very same experiments, yet with time rather than training rounds. Confirming the results we give concerning the throughput of the compared systems, Figure 11a shows that the vanilla deployment, i.e., TensorFlow converges faster than the crash–tolerant protocol, which converges faster than the Byzantine–resilient protocols. In addition, AggregaThor is faster than Garfield. Figure 11b sheds even more light on the cost of fault–tolerance in general. For instance, although Figure 4b (in the main manuscript) shows that the crash–tolerant protocol can reach the same final accuracy as the vanilla deployment, i.e., PyTorch, Figure 11b shows that such an accuracy would be reached (by the crash-tolerant protocol) in more than 3x time than the vanilla deployment. Such a figure shows that the cost of tolerating mere crash failures is not negligible. Furthermore, it emphasizes the accuracy loss introduced by the Byzantine–resilient deployment, i.e., Garfield. Finally, we can observe from this figure that the cost of Byzantine–resilience, compared to crash–resilience, is not big, in terms of throughput.
2 Convergence with MDA as a GAR

In this section, we show the convergence of GARFIELD’s protocol while using MDA as a GAR. We use the CPU–based cluster and the TensorFlow–based implementation in this experiment. We use the same setup as in Section 5.2 (with the default setup of TensorFlow–based experiments) in the main paper.

Figure 12(a) shows that all compared systems achieve almost the same convergence rate. Moreover, it shows that deploying the Byzantine variant of GARFIELD does not add any overhead compared to the correct one (vanilla TensorFlow), in terms of convergence steps. However, the cost of resilience appears in Figure 12(b) which depicts the convergence rate against time. For example, vanilla TensorFlow reaches accuracy of 60% in 364 seconds, which is 15% better than the crash–tolerant deployment, where the Byzantine deployment reaches the same accuracy level in 23% more time (which is the cost of Byzantine–resilience) compared to the latter.

3 Effect of The Number of Byzantine Machines

In this section, we show the effect of changing the number of both Byzantine workers and Byzantine servers on the performance of GARFIELD. Our metric here is throughput and we use the same setup as the throughput experiments in the main manuscript. We run experiments on both CPUs (using TensorFlow) and GPUs (using PyTorch).

Figure 13: Throughput of Garfield with different number of Byzantine workers.
**Number of Byzantine workers.** As increasing the number of Byzantine workers ($f_w$) does not call for increasing the total number of workers, we fix $n_w$ and hence, fixing the effective batch size in all cases. Increasing $f_w$ leads to increasing the number of replies which the servers wait for before proceeding to the aggregation phase. Thus, a higher value of $f_w$ slightly decreases the throughput, especially in the presence of stragglers. This is confirmed in Figure 13 with $f_w = 3$ for running on both CPUs and GPUs. An interesting fact to note here is that increasing $f_w$ does not affect the number of iterations required for convergence since GARFIELD uses replies from a higher number of workers when increasing $f_w$ (e.g., $2f_w + 3$).

![Figure 14: Throughput of Garfield with different number of Byzantine servers.](image)

**Number of Byzantine servers.** Increasing the number of Byzantine servers ($f_{ps}$) calls for increasing the total number of the server’s replicas ($n_{ps}$) so as to follow the Byzantine–resilience condition: $n_{ps} \geq 3f_{ps} + 3$ or $n_{ps} \geq 2f_{ps} + 3$ in case of using a synchronous network. Increasing $f_{ps}$ introduces new communication links, which leads to throughput drop as shown in Figure 14. This throughput drop is confirmed in the *state machine replication* (SMR) literature [35, 6], and we believe that the amount of drop (less than 45% in our case) is still reasonable compared to what is reported before in the literature of SMR [19]. To even confirm more that this drop completely depends on the increased communication messages, we can see the same behavior (with approximately the same degradation ratio) with GPU (Figure 14b) as with CPUs (Figure 14a).

## 4 Throughput in PyTorch

This section gives more results of GARFIELD implementation in PyTorch. Specifically, we show the performance of GARFIELD while training different models and dissect such overhead to understand its reasons.

**Model dimension.** In this section, we show the performance (basically throughput slowdown with respect to vanilla PyTorch) of our PyTorch–based implementation of GARFIELD, while training several models. We use the same models as in Figure 5 in the main manuscript (except for ResNet-200 which we replace with ResNet-152 as the former is not included in the *TorchVision* models) and we use our GPU–based cluster to carry out this experiment. Figure 15 shows similar results to its twin (using GPUs) in the main manuscript (Figure 5b). It is emphasized in this figure that the
cost of fault–tolerance is not clear with training small networks, i.e., MNIST_CNN and CifarNet. In addition, the cost of Byzantine resilience compared to crash resilience is moderate. Interestingly, our PyTorch implementation of GARFIELD achieves higher slowdown compared to its vanilla competitor, i.e., PyTorch than what is achieved by the TensorFlow–based implementation (Figure 5b, main manuscript). The main reason for this slowdown is the efficiency of the communication backend that is used by PyTorch, especially when running on GPUs. Vanilla PyTorch uses the reduce() abstraction which (1) uses GPU-to-GPU communication and (2) calculates the average of the gathered vectors immediately when received. Conversely, the GARFIELD communication backend uses gather() to collect the input vectors, and the receiving end, be it a worker or a server, waits for all replies before applying the chosen GAR. The performance difference between both abstractions is emphasized more with the big models. On the TensorFlow side, both the vanilla deployment and the GARFIELD implementation uses gRPC so, the performance difference comes only from the difference in the number of communication rounds and the number of messages transmitted in both deployments.

Overhead breakdown. In this section, we dissect the performance of GARFIELD on PyTorch to understand the reasons for the overhead. Figure 16 shows the time spent per iteration on computation, communication, and aggregation (if any). First, we can see that the fault–tolerant systems spend less time in computation than in vanilla PyTorch. This is because some of the computation done by the GARFIELD–enabled systems is hidden in the communication (due to our pipelined design of GARFIELD). The vanilla PyTorch deployment has the lowest communication overhead compared to the fault–tolerant alternatives. This is because it uses the fast reduce() networking abstraction which enables GPU-to-GPU communication and processes the data.
as they come immediately (using averaging). As our design of GARFIELD (on PyTorch) pipelines communication with aggregation (as mentioned in Section 4.3 in the main manuscript), Figure 16 shows the contribution of both in one orange–and–green bar. Such a combined bar is higher in GARFIELD (the Byzantine–tolerant deployment) than the crash–tolerant deployment for two reasons. First, GARFIELD requires more communication rounds and even more messages per round compared to the crash–tolerant algorithms. Though we optimize the many-to-many communication (e.g., servers-to-workers) by parallelizing communication links and using abstractions that allow GPU-to-GPU communication, the crash–tolerant deployment can still benefit from its less demanding behavior to the network. Second, the crash–tolerant deployment uses the fast Average rule for gradient aggregation and does not use any rule for aggregating models, however, GARFIELD uses more complex rules, e.g., Multi–Krum and MDA to aggregate both gradients and models.

5 Parameter Vectors Alignment

We do some micro–measurements to observe the alignment of the parameter vectors. In this section, we describe our methodology and results.

Methodology. We consider the quantities $\theta_i^{(i)}, \forall i \in [1..n−f]$ and $t > t_s$ (our assumption must hold eventually, e.g., after some large number of steps $t_s$). First, we calculate the differences between all parameter vectors (which we call difference vectors) and kept ones with the $k$ highest norms. Then, we calculate the angle between these difference vectors to see how they are aligned. We do that every 20 steps, throughout the training procedure, to empirically check whether the considered assumption holds or not.

Results. Generally, we find that the angle between differences (from difference vectors) of the highest norms is always close to 0°. We give here the values of $\cos(\phi)$ where, $\phi$ is the angle between two difference vectors which are $a$ and $b$. This is given by $\frac{a \cdot b}{\|a\|\|b\|}$. Thus, the closer this value to 1, the closer the angle to 0°. Sample results are given in Table 2.

Table 2: Experimental results while studying contraction of parameter vectors. It gives for some steps the biggest 2 norms of differences between parameter vectors collected by correct parameter servers along with $\cos(\phi)$ where $\phi$ is the angle between these two difference vectors. This is recorded after some large step number.

| Step  | $\cos(\phi)$    | max_diff$_1$ | max_diff$_2$ |
|-------|-----------------|--------------|--------------|
| 1340  | 0.9822574257850647 | 1.4122562    | 1.4163861    |
| 1380  | 0.9926297664642334 | 1.3927394    | 1.3937825    |
| 1400  | 0.9881128072738647 | 1.493591     | 1.5035304    |
| 1440  | 0.9863847494125366 | 1.345111     | 1.3537675    |
| 1480  | 0.9819352030754089 | 1.3270174    | 1.3435347    |