Implementation of Secure Byzantine Robust Machine Learning[1] Using GARFIELD [2]
Internship project@ DCL

Sahar Sheikholeslami
Email : seslami2@gmail.com

***Abstract*— Garfield [2] is a library for distributed machine learning, specifically designed for the fully byzantine environment. A broad variety of byzantine robust ML algorithms can be implemented using this framework. In this work, we add more features to this framework to make it compatible with new applications. The most notable added features are security, and variety in worker/server. We implement an application [1] to test these new features and also report on experimental evaluations of this application. Our source code is publicly available at : github.com**

I. INTRODUCTION

Billions of new data are shared every day on the internet. The amount of data generated keeps increasing continuously and has been an excellent opportunity for Machine learning algorithms. Yet, precisely because of the huge amount of data available, these algorithms require immense demands of computing resources in order to train the ML model and provide accurate predictions.

Distributing Machine Learning (ML) tasks seems to be the only way to cope with this ever-growing datasets [3][4]. The now classical parameter server architecture is a common way to distribute the learning task[5][6]. In short, a central server holds the model parameters (e.g., weights of a neural network) while a set of workers perform the back-propagation computation[7], typically following the standard optimization algorithm: stochastic gradient descent (SGD)[8], on their local data, using the latest model they pull from the server. This server in turn gathers the updates from the workers in the form of gradients and aggregates them, usually through averaging[9]. However, this scheme is very fragile because averaging does not tolerate a single corrupted input[10], while the diversity of machines increases the probability of misbehavior somewhere in the network.

A classical approach to mask failures in distributed systems is to use a state machine replication protocol[11], but some theoretical approaches have been recently proposed to address Byzantine-resilience without replicating the workers[10][12]. In short, the idea is to use more sophisticated forms of aggregation (e.g. median) than simple averaging.

Besides scaling, another motivation for distributed ML schemes is privacy. The workers could be user machines keeping their data locally and collaborating through a parameter server to achieve some global machine learning task [13]. However, all the aggregation rule which was mentioned before does not preserve the input privacy because the server observes the models directly. The input privacy requires each party to learn nothing more than the output of computation which in this paradigm means the aggregated model updates. Although many works have been done in tackling robustness, privacy, or security individually, their combination has rarely been studied. In [1], a secure two-server protocol was proposed that offers both input privacy and Byzantine-robustness. In addition, this protocol is communication-efficient, fault-tolerant, and enjoys local differential privacy.

Most work on Byzantine resilient ML has however been theoretical and, it is not clear how to put the published algorithms to work, especially in the pragmatic form of library extensions to existing, and now classical, ML frameworks, namely TensorFlow [14] and PyTorch [15]. These frameworks share two specific characteristics that go against Byzantine resilience. First, and for performance reasons, they rely on a shared memory design. For instance, TensorFlow uses one shared computation graph among all machines to define the learning pipeline. Such a design is problematic as Byzantine nodes can corrupt the learning state at honest ones. Second, most of the high–level communication abstractions given by such frameworks assume trusted, highly–available machines. For instance, the distributed library of PyTorch allows for collective communication among processes, yet such calls block indefinitely in case of a process crash or network failure.

Recently two system support for the byzantine robust machine learning algorithm, namely AGGREGATHOR [16] and GARFIELD[2], were proposed. AGGREGATHOR is a scalable ML system that achieves Byzantine resilience, yet only for Byzantine workers. It is built on TensorFlow and supports training only on CPUs. AGGREGATHOR uses one central, trusted server while tolerating Byzantine workers. Same as AGGREGATHOR, Garfield is a library that enables the development of Byzantine ML applications on top of TensorFlow and PyTorch while achieving transparency: applications developed with either framework do not need to change their interfaces to tolerate Byzantine failures. GARFIELD relies on an object-oriented design, which makes it possible to adapt the components in the network and how they interact, according to the desired Byzantine resilient scheme. In particular, GARFIELD allows for both synchronous and asynchronous communication. GARFIELD

also includes several statistically robust gradient aggregation rules (GARs), which can filter out Byzantine replies' effect. GARFIELD also supports full-stack computations on both CPUs and GPUs. unlike AGGREGATHOR, Garfield supports three primary distributed ML settings: (a) A single trusted central server and multiple workers, (b) Multiple servers and workers, and (c) decentralized peer-to-peer setting with no distinction between servers and workers.

Contributions. In some byzantine resilience algorithms [1] having more than one server is not because of the possibility of failure in them, but the reason is that they aggregate gradients and update the model collaboratively, so different algorithm servers would perform different algorithms. Implementing this algorithm using GARFIELD is not possible because in all server/worker architecture in GARFIELD, servers are assumed identical, and they all follow the same algorithm. It is also true for workers. One fundamental mandatory change in GARFIELD for supporting [1] algorithm is the ability to defines different servers.

One of the very primary assumptions in machine learning with privacy-preserving is that all communication channels should be secure, while all communication channels are insecure. Making all the communications in Garfield secure is another compulsory change needed for implementing a secure machine learning algorithm with this library.

A GAR is merely a function of $(R^d)^q \rightarrow R^d$, with d, the dimension of the input vector space (R^d) (i.e., a gradient or a model), and q, the number of input vectors to be aggregated. These GARs' inputs are q gradients, and the output is a vector with special statistical properties that make them safe to use in the Byzantine setting. Input privacy requires keeping the gradient private, so instead of using a GAR whose input is gradient, distance-based robust aggregation rules can be applied [10][17], with gradients distances as an input, not gradients. Supporting distance-based aggregation rules with gradients distances as an input is the last change Garfield needs to be able to secure byzantine robust machine learning [1].

II. GARFIELD

This section presents Garfield's architecture details and explains the potential applications that could be implemented using this framework easily.

A. System Component:

Fig 1 shows a high-level view of the GARFIELD architecture.

1) Network: GARFIELD relies on gRPC for point-to-point communication and uses a Protocol buffer for serializing data.

As in many RPC systems, gRPC is based on defining a service, specifying the methods that can be called remotely with their parameters and return types. gRPC usually comes with protocol buffers as its underlying message interchange format. Protocol buffers are a fast, small and straightforward extensible mechanism for serializing structured data.

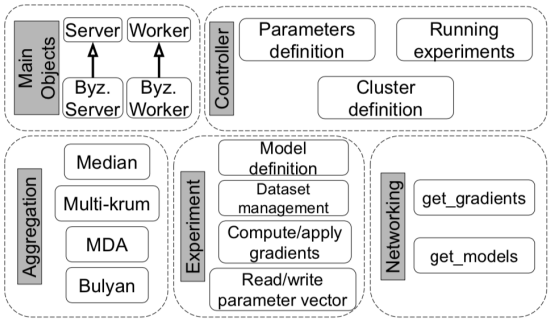


Fig. 1: The GARFIELD system components

GARFIELD uses a pull-based model for transferring data, meaning that if a node needs some data, it requests that data from other nodes using remote procedure calls. Each worker and server has a gRPC server to handle these requests.

GARFIELD provides three different protocol buffers: request, model, and gradient. it also defines two requests for transferring data: get_gradient() and get_model(). The former is applied for getting the computed gradients by the workers. It gets the request protocol buffer as an input, containing the current iteration index and the number of workers. The latter abstraction works in the same way yet fetches models from servers instead of gradients from workers.

All the communication between different nodes should be done using these requests. In order to reduce communication time, The replicated communication between workers and servers for requesting gradients and updating models is parallelized.

2) Main Object: Same as the famous parameter server model, Garfield's two main objects are parameter server and worker. These two main objects are both inherited from an abstract class, Server. The parameter server is in charge of storing and updating the model in each iteration, while workers train this model by computing gradient on their local data. For updating the model in each iteration, the parameter server demands workers to compute a gradient estimate based on the model on the same iteration. For this, the server use get-gradient(), which is described in detail before. Workers need to pull the renewed model in each iteration, which has been done through get_model(). This module is also used in algorithms requiring servers to exchange their model's states, i.e., multiple servers and workers. In addition to the Networking methods, the server object reveals methods to update the model state, given some optimizer and a gradient estimate, re-write the model, which is useful in the case of multiple server replicas, and Compute accuracy, given the model state and a test set. Each worker owns some data and defines a loss function; its goal is to respond to the parameter server request by computing a gradient estimate using its data. GARFIELD defines two other objects for supports byzantine behavior: Byzantine Server and Byzantine Worker, which inherit from the main objects, Server and Worker, respectively. Different attacks such as reversing vector, dropping vector, random

vectors, and fall of empires are implemented and could be used by either byzantine object.

3) Aggregation: Once a server gets all the gradients, it needs to aggregate them. GARFIELD supports four different Byzantine–resilient GARs on both CPU and GPU:

1) **Median**[18] computes the coordinate-wise median among the input gradients and outputs one gradient of these medians. Median requires $q \geq 2f + 1$.
2) **Krum**[10] assigns a score to each gradient (based on a sum of distances with the closest neighbors), and then returns the smallest scoring gradient.
3) **MDA**[19] 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.
4) **Bulyan**[20] robustly aggregates q gradients by iterating several (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 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.

a Wrappers is implemented in order to use as custom operations in TensorFlow and PyTorch. With the help of this wrapper, both PyTorch and TensorFlow use the same interfaces.

4) Controller: This module is responsible for controlling the training task. It is used for cluster deployment, the definition of parameters, as well as for launching experiments. This encompasses parsing the cluster information, such as nodes' jobs (servers or workers) and 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.

5) Experiment: This module abstracts the available models and datasets for training. Our design gives a unified interface to the TensorFlow slim research models, Keras, and TorchVision models. This enables us to experiment with various models, e.g., ResNet and VGG. We leverage the compute and apply gradients functions to the underlying system on the training side, be it TensorFlow or PyTorch.

B. Application

GARFIELD supports three different ML settings. For testing and checking GARFIELD's different functionality and evaluate its performances through this application. For one central trusted server and several workers settings, AGGRATHOR [16] is implemented. for several servers and workers, ByzSGD [12] is implemented, which is the first Byzantine-resilient machine learning protocol to tolerate Byzantine servers as well as Byzantine workers in the parameter server architecture. Essentially, ByzSGD replicates the server on multiple machines and relies on communication among the server replicas to (roughly speaking) agree on the same model. Learn [] considers a fully decentralized setup: a bunch of devices/workers collaborate to train a model without a central server. Such devices communicate in a peer-to-peer fashion in each training step. LEARN is the first protocol to solve the decentralized learning problem in the non-iid setup: the data is not assumed to be identically nor independently distributed among the workers. Learn is a peer-to-peer application of GARFIELD.

III. Algorithm

In this paper, a secure byzantine robust aggregation framework was proposed.

A. problem setup

In this setting, n workers, with the help of two servers, are trying to train a shared model collaboratively. Each worker has its private part of the training dataset and computes its private model update (e.g., a gradient based on its data) denoted by the vector x_i. The aggregation protocol aims to calculate the aggregation, which is then used to update a public model. While the result z is shared in all cases, the protocol must keep each x_i private from any adversary or other workers.

1) Security model: Servers are assumed honest but curious, which do not collude with each other, but server-worker collusion is allowed. The algorithm guarantees the strong notion of input privacy, which means the servers and workers know nothing more about each other than what can be reached from the public aggregate result.

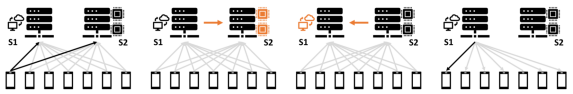
2) Byzantine robustness model: The algorithm supports the Byzantine worker model, which assumes that workers can send arbitrary adversarial messages trying to compromise the process. The algorithm assumes that less than half of the workers can be byzantine.

3) Additive secret sharing: Secret sharing is a way to split any secret into multiple parts such that no part leaks the secret. Formally, suppose a scalar a is a secret and the secret holder shares it with k parties through secret-shared values $\langle a \rangle$. In this paper, we only consider additive secret-sharing where $\langle a \rangle$ is a notation for the set $\{a_i\}_i = 1^k$ which satisfy $a = \sum_{p=1}^{k} a_p$, with a_p held by party p. Crucially, it must not be possible to reconstruct a from any a_p. For vectors like x, their secret-shared values $\langle x \rangle$ are simply component-wise scalar secret-shared values.

4) Two-server setting: As explained before, two non-colluding servers are assumed: model server(S1) and worker server(S2). The server model is responsible for storing the public model and holds the output of each aggregation. The worker server holds intermediate values to perform byzantine aggregation. Another key assumption is that the servers have no incentive to collude with workers, perhaps enforced via a huge potential penalty if exposed. It is realistic to assume that the communication link between the two servers S1 and S2, is faster than the individual links to the workers. To perform robust aggregation, the servers

will need access to a sufficient number of Beaver's triples. These are data-independent values required to implement secure multiplication in MPC on both servers, and can be precomputed beforehand.

B. Robust secure aggregation



(a) **WorkerSecretSharing**: each worker i secret-shares its update x_i locally and uploads them to **S1** and **S2** separately.

(b) **RobustWeightSelection**: Compute and reveal $\{\|x_i - x_j\|^2\}_{i<j}$ on **S2** and select a robust set of indices represented by $p = \{p_i\}_i$ by calling the Byzantine-robust oracle.

(c) **AggregationAndUpdate**: Compute and reveal aggregation $z = \sum_{i=1}^{n} p_i x_i$ on **S1**. **S1** updates the public model.

(d) **WorkerPullModel**: Each worker i pulls model from **S1**.

Fig. 2: Illustration of Algorithm 1. The orange components on servers represent the computation- intensive operations at low communication cost between servers.

Algorithm 1 Two-Server Secure Robust Aggregation (Distance-Based)

Setup: n workers, αn of which are Byzantine. Two non-colluding servers **S1** and **S2**.
Workers: (WorkerSecretSharing)
1. split private x_i into additive secret shares $\langle x_i \rangle = \{x_i^{(1)}, x_i^{(2)}\}$ (such that $x_i = x_i^{(1)} + x_i^{(2)}$)
2. send $x_i^{(1)}$ to **S1** and $x_i^{(2)}$
Servers:
1. $\forall i$, **S1** collects gradient $x^{(1)}$ and **S2** collects $x^{(2)}$
2. **(RobustWeightSelection)**:
   (a)
   for (x_i, x_j) compute their Euclidean distance **do**
   i) On **S1** and **S2**, compute $\langle x_i - x_j \rangle = \langle x_i \rangle - \langle x_j \rangle$ locally
   ii) Use precomputed Beaver's triples to compute the distance $\| x_i - x_j \|$
   end for
   (b) **S2** perform robust aggregation rule $p = Agg$ $(\| x_i - x_j \|^2_{i<j})$
   (c) **S2** secret-shares $\langle p \rangle$ with **S1**
3. **(AggregationAndUpdate)**:
   (a) On **S1** and **S2**, use MPC multiplication to compute $\langle \sum_{i=1}^{n} p_i x_i \rangle$
   (b) **S2** sends its share of $\langle \sum_{i=1}^{n} p_i {x_i}^{(2)} \rangle$ to **S1**
   (c) **S1** reveals $z = \langle \sum_{i=1}^{n} p_i {x_i}^{(2)} \rangle$ to all workers.
WORKERS:
1. **(WorkerPullModel):** Collect z and update model locally

Each worker first splits its private vector x_i into two additive secret shares, and transmits those to each corresponding server, ensuring that neither server can reconstruct the original vector on its own. The two servers then execute our secure aggregation protocol. On the level of servers, the protocol is a two-party computation (2PC). Then the two servers exactly emulate an existing Byzantine robust aggregation rule, at the cost of revealing only distances of worker gradients on the server (the robust algorithm is presented in Algorithm 1). Finally, the resulting aggregated output vector z is sent back to all workers and applied as the update to the public machine learning model. How Algorithm 1 replaces the aggregation with any distance-based robust aggregation rule **Aggr** is explained below. The key idea is to use two-party MPC to securely compute multiplication.

- **WorkerSecretSharing** (Fig 2a): As before, each worker i secret shares $\langle x_i \rangle$ distributed over the two servers **S1** and **S2**.
- **RobustWeightSelection** (Fig 2b): After collecting all secret-shared values $\{\langle x_i \rangle\}_i$, the servers compute pairwise difference $\{\langle x_i - x_j \rangle\}_{i<j}$ locally. **S2** then reveals to itself exclusively in plain text all of the pairwise Euclidean distances between workers $\{\| x_i - x_j \|^2\}_{i_j}$ with the help of precomputed Beaver's triples The distances are kept private from **S1** and workers. **S2** then feeds these distances to the distance- based robust aggregation rule **Aggr**, returning (on **S2**) a weight vector $p = \{p_i\}_{i=1}^n$ (a selected subset indices can be converted to a vector of binary values), and secret-sharing them with **S1** for aggregation.
- **AggregationAndUpdate** (Figure 1c): Given weight vector p from previous step, we would like **S1** to compute $\sum_{i=1}^{n} p_i x_i$. **S2** secret shares with **S1** the values of $\{\langle p_i \rangle\}$ instead of sending in plain-text since they may be private. Then, **S1** reveals to itself, but not to **S2** , in plain text the value of $z = sum_{i=1}^{n} p_i x_i$ using secret-shared multiplication and updates the public model.
- **WorkerPullModel** (Figure 2d): Workers pull the latest public model on **S1** and update it locally.

IV. Implementation

In this section, the different part added to GARFIELD is explained. It is important to note that all the changes in the library must be done while other implemented applications are still runnable without any changes.

A. security

In algorithm 1, S1 pulls its own secret shares of gradients, so does S2. So besides Authentication, Authentication is needed for this algorithm. Every request should be carefully authenticated, and if the sender has permission for what it requested, then the request performs. As it is explained before, GARFIELD relies on gRPC for point–to–point pull-based communication. gRPC provided some modules which helped to make the connection secure.

Every node has an SSL certificate that indicates who it is and its role. The algorithm proposed three different roles: model server(S1), worker Server(S1), and worker. A Trusted third party is needed for play as Certificate Authority(CA) and signs all certificates.

For implementing authentication, each client used secure_connection() instead of insecure _connect(), which is used already in GARFIELD. secure_connection() needs SSL

credentials as a parameter, and SSL credentials consist of the client's private key, CA certificate, and client's certificate. Each node has the root certificate, and before accepting to start a connection, check the credential and authenticate the connection.

For authorization, gRPC provides an interceptor for the gRPC server. Interceptor receives the requests and checks whether the sender has access to what it requests or not.

Adding security is expensive, and other applications might not be interested in using it, so in controller modules,secure_channel must be set to true if the application wants a secure connection.

B. Differentiate the server

Garfield supports three different distributed ML settings:

- A trusted server with workers
- Several workers and several servers
- Peer-to-peer setting

Although algorithm 1 assumes trusted servers, the number of servers is not matching with the first one. Second architecture, all servers are considered to be identical and performing the same algorithm. Apart from that, it's the best one among the supported architectures for algorithm 1. All nodes in Garfield have a gRPC server, which clarifies how to respond to different requests. Instead of using the same gRPC server for all servers, it can be sent as a parameter to a parameter server or workers. Request and message format in Garfield also is very restricted. In the algorithm 1, more data than gradient and model is exchanged(.e.g. pairwise distances). This data exchange a gradient and should be serialized.

In total, three new objects were added to Garfield for this application: SecureWorker, WorkerServer, and ModelServer. SecureWorks and ModelSever are inherited from Worker and PS, respectively.

V. Experiments

An image classification task is considered for the experiment due to its wide adoption as a benchmark for distributed ML systems, e.g. [3].

For the datasets, MNIST is employed. It consists of 70,000 handwritten digits; each has a size of 28*28. A small convolutional neural network (CNN) is used for neural network architectures, training less than 100k parameters. the mini-batch size is 500 is all the expriments. All the simulation is run on a local machine. In algorithm 1, the communication link between two servers is faster than worker-server links. This fact is considered in the experiments. Fig depicts the cost of security in terms of running time.

In Fig 3 AGGREGATHOR with unsecured links, one of GARFIELD's applications, is compared to the case that all the communication is secure. the y-axis represents the running time after 1000 iteration.

Fig 4 shows the AGGREGATHOR and algorithm 1 have almost the same convergence behavior. The experiment is run with six normal workers and a byzantine worker for both deployments.

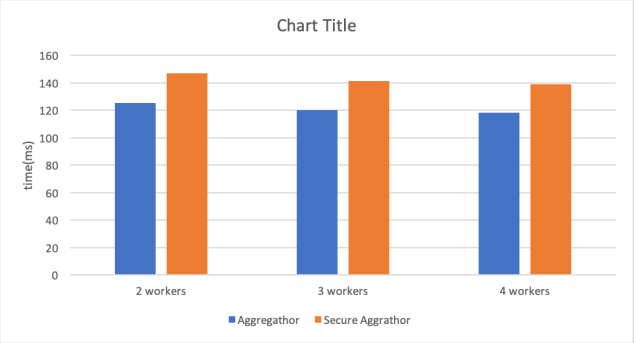


Fig. 3: overhead of security

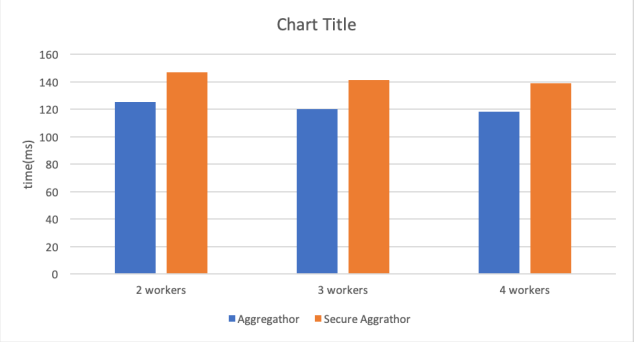


Fig. 4: Convergence in the presence of Byzantine workers.

VI. Future work

GARFIELD, a byzantine resilience library, supports multiple statistically–robust gradient aggregation rules (GARs), which can be combined in various ways for different resilience properties.

For making GARFIELD more powerful, it should be updated frequently with state-of-the-art robust aggregation rules and attacks. Besides that, asynchronous robust aggregation rules[21] has been recently proposed, which are not supported by GARFIELD.

References

[1] S. P. K. Lie He and M. Jaggi, "Secure byzantine-robust machine learning," *arXiv:2006.04747*.

[2] J. Z. A. J. D. M. D. S. G. G. I. M. I. e. a. MartınAbadi, PaulBarham, "Garfield: System support for byzantine machine learning," *OSDI*.

[3] J. A. Trishul M Chilimbi, Yutaka Suzue and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system,"

[4] B. Y. E. S. S. V. D. L. J. F. D. T. M. A. S. O. e. a. Xiangrui Meng, Joseph Bradley, "Mllib: Machine learning in apache spark," *JMLR*.

[5] Z. Y. e. a. Mu Li, Li Zhou, "Parameter server for distributed machine learning," *Big Learning NIPS Workshop*.

[6] J. W. P. A. J. S. A. A. V. J. J. L. E. J. S. Mu Li, David G Andersen and B.-Y. Su, "Scaling distributed machine learning with the parameter server," *Elsevier*.

[7] R. Hecht-Nielsen, "Theory of the backpropagation neural network. in neural networks for perception," *OSDI*.

[8] G. E. H. David E Rumelhart and R. J. Williams, "Learning representations by back-propagating errors," *nature*.

[9] B. M. Jakub Konecny‘ and D. R. . F. optimization, "Distributed optimization beyond the datacenter," *arXiv preprint*.

[10] R. G. Peva Blanchard, El Mahdi El Mhamdi and J. Stainer., "Machine learning with adversaries: Byzantine tolerant gradient descent," *Neural Information Processing Systems*.

[11] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *CSUR*.
[12] A. G. L. N. H. El-Mahdi El-Mhamdi, Rachid Guerraoui and S. bastien Rouault, "Genuinely distributed byzantine machine learning," *Proceedings of the 39th Symposium on Principles of Distributed Computin*.
[13] C.-A. G. I. M. H. B. M. I. T. K. Abadi, M. and L. Zhang, "Deep learning with differential privacy.," *SIGSAC*.
[14] J.-Z. A. J. D. M. D. S. G. G. I. M. I. e. a. MartınAbadi, PaulBarham, "Tensorflow: A system for large-scale machine learning," *OSDI*.
[15] F. M. A. L. J. B.-b. G. C. T. K. Z. L. N. G. L. A. e. a. Adam Paszke, Sam Gross, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*.
[16] R. G. A. G. Georgios Damaskinos, El Mahdi El Mhamdi and S. Rouaul, "Aggregathor: Byzantine machine learning via robust gradient aggregation," *SysML*.
[17] S. K. Cong Xie and I. Gupta, "Fall of empires: Breaking byzantine-tolerant sgd by inner product manipulation," *arXiv preprint*.
[18] O. K. Cong Xie and I. Gupta, "Generalized byzantine-tolerant sgd," *arXiv preprint*.
[19] P. J. Rousseeuw, "The hidden vulnerability of distributed learning in byzantium," 1985.
[20] R. G. El Mahdi El Mhamdi and S. Rouault, "Multivariate estimation with high breakdown point. mathematical statistics and applications," *International Conference on Machine Learning*.
[21] E. M. E. M. G. R. P. R.-T. M. e. a. Damaskinos, G., "Asynchronous byzantine machine learning (the case of sgd)," *ICML*.