Fully Distributed Privacy Preserving Mini-Batch Gradient Descent Learning*

Gábor Danner and Márk Jelasity

University of Szeged, and MTA-SZTE Research Group on AI, Hungary
{danner,jelasity}@inf.u-szeged.hu

Abstract. In fully distributed machine learning, privacy and security are important issues. These issues are often dealt with using secure multiparty computation (MPC). However, in our application domain, known MPC algorithms are not scalable or not robust enough. We propose a light-weight protocol to quickly and securely compute the sum of the inputs of a subset of participants assuming a semi-honest adversary. During the computation the participants learn no individual values. We apply this protocol to efficiently calculate the sum of gradients as part of a fully distributed mini-batch stochastic gradient descent algorithm. The protocol achieves scalability and robustness by exploiting the fact that in this application domain a "quick and dirty" sum computation is acceptable. In other words, speed and robustness takes precedence over precision. We analyze the protocol theoretically as well as experimentally based on churn statistics from a real smartphone trace. We derive a sufficient condition for preventing the leakage of an individual value, and we demonstrate the feasibility of the overhead of the protocol.

Keywords: fully distributed learning, mini-batch stochastic gradient descent, P2P smartphone networks, secure sum

1 Introduction

Our long-term research objective is to design fully distributed machine learning algorithms for various distributed systems including networks of smartphones, smart meters, or embedded devices. The main motivation for a distributed solution in our cloud-based era is to preserve privacy by avoiding the central collection of any personal data. Another advantage of distributed processing is that this way we can make full use of all the local personal data, which is impossible in cloud-based or private centralized data silos that store only specific subsets of the data.

In our previous work we proposed several distributed machine learning algorithms in a framework called gossip learning. In this framework models perform random walks over the network and are trained using stochastic gradient descent [18] (see Section 4). This involves an update step in which nodes use their local data to improve each model they receive, and then forward the updated model along the next step of the random walk. Assuming the random walk is secure—which in itself is a research problem on

---

its own, see e.g. [13]—it is hard for an adversary to obtain the two versions of the model right before and right after the local update step at any given node. This provides reasonable protection against uncovering private data.

However, this method is susceptible to collusion. If the nodes before and after an update in the random walk collude they can recover private data. In this paper we address this problem, and improve gossip learning so that it can tolerate a much higher proportion of honest but curious (or semi-honest) adversaries. The key idea behind the approach is that in each step of the random walk we form groups of peers that securely compute the sum of their gradients, and the model update step is performed using this aggregated gradient. In machine learning this is called mini-batch learning, which—apart from increasing the resistance to collusion—is known to often speed up the learning algorithm as well (see, for example, [8]).

It might seem attractive to run a secure multiparty computation (MPC) algorithm within the mini-batch to compute the sum of the gradients. The goal of MPC is to compute a function of the private inputs of the parties in such a way that at the end of the computation, no party knows anything except what can be determined from the result and its own input [24]. Secure sum computation is an important application of secure MPC [7].

However, we do not only require our algorithm to be secure but also fast, lightweight, and robust, since the participating nodes may go offline at any time [2] and they might have limited resources. One key observation is that for the mini-batch algorithm we do not need a precise sum; in fact, the sum over any group that is large enough to protect privacy will do. At the same time, it is unlikely that all the nodes will stay online until the end of the computation. We propose a protocol that—using a tree topology and homomorphic encryption—can produce a "quick and dirty" partial sum even in the event of failures, has adjustable capability of resisting collusion, and can be completed in logarithmic time.

2   Related Work

There are many approaches that have goals similar to ours, that is, to perform computations over a large and highly distributed database or network in a secure and privacy preserving way. Our work touches upon several fields of research including machine learning, distributed systems and algorithms, secure multiparty computation and privacy. Our contribution lies in the intersection of these areas. Here we focus only on related work that is directly relevant to our present contributions.

Algorithms exist for completely generic secure computations, Saia and Zamani give a comprehensive overview with a focus on scalability [22]. However, due to their focus on generic computations, these approaches are relatively complex and in the context of our application they still do not scale well enough, and do not tolerate dynamic membership either.

Approaches targeted at specific problems are more promising. Clifton et al. propose, among other things, an algorithm to compute a sum [7]. This algorithm requires linear time in the network size and it does not tolerate node failure either. Bickson et al. focus on a class of computations over graphs, where the computation is performed in an iterative manner through a series of local updates [3]. They introduce a secure algorithm

to compute local sums over neighboring nodes based on secret sharing. Unfortunately, this model of computation does not cover our problem as we want to compute mini-batches of a size independent of the size of the direct neighborhood, and the proposed approach does not scale well in that sense. Besides, the robustness of the method is not satisfactory either [17]. Han et al. address stochastic gradient search explicitly [12]. However, they assume that the parties involved have large portions of the database, so their solution is not applicable in our scenario.

The algorithm of Ahmad and Khokhar is similar to ours [1]. They also use a tree to aggregate values using homomorphic encryption. In their solution all the nodes have the same public key and the private key is distributed over a subset of elite nodes using secret sharing. The problem with this approach in our application is that for each mini-batch a new key set has to be generated for the group, which requires frequent access to a trusted server, otherwise the method is highly vulnerable in the key generation phase.

We need to mention the area of differential privacy [9], which is concerned with the the problem that the (perhaps securely computed) output itself might contain information about individual records. The approach is that a carefully designed noise term is added to the output. Gradient search has been addressed in this framework (for example, [20]). In our distributed setup, this noise term can be computed in a distributed and secure way [10].

3 Model

Communication. We model our system as a very large set of nodes that communicate via message passing. At every point in time each node has a set of neighbors forming a connected network. The neighbor set can change over time, but nodes can send messages only to their current neighbors. Nodes can leave the network or fail at any time. We model leaving the network as a node failure. Messages can be delayed up to a maximum delay. Messages cannot be dropped, so communication fails only if the target node fails before receiving the message.

The set of neighbors is either hard-wired, or given by other physical constraints (for example, proximity), or set by an overlay service. Such overlay services are widely available in the literature and are out of the scope of our present discussion. It is not strictly required that the set of neighbors are random, however, we will assume this for the sake of simplicity. If the set is not random, then implementing a random walk with a uniform stationary distribution requires additional well-proven techniques such as Metropolis-Hastings sampling or structured routing [23].

Data distribution. We assume a horizontal distribution, which means that each node has full data records. We are most interested in the extreme case when each node has only a single record. The database that we wish to perform data mining over is given by the union of the records stored by the nodes.

Adversarial model. We assume that the adversaries are honest but curious (or semi-honest). That is, nodes corrupted by an adversary will follow the protocol but the adversary can see the internal state of the node as well as the plaintext of the messages that the node receives or sends. The goal of the adversary is to learn about the private data

of other nodes (note that the adversary can obviously see the private data on the node it observes directly).

We assume a static adversarial model, which means that the corrupted nodes are picked a priori, independently of the state of the protocol or the network. As of the number of corrupted nodes, we will consider the threshold model, in which at most a given number of nodes are corrupted, as well as a probabilistic model, in which any node can be corrupted with a given constant probability [16].

Wiretapping is assumed to be impossible. In other words, communication channels are assumed to be secure. This can easily be implemented if there is a public key infrastructure in place.

We also assume that adversaries are not able to manipulate the set of neighbors. In each application domain this assumption translates to different requirements. For example, if an overlay service is used to maintain the neighbors then this service has to be secure itself.

4   Background on Gossip Learning

Although not strictly required for understanding our key contribution, it is important to briefly overview the basic concepts of stochastic gradient descent search, and our gossip learning framework (GOLF) [18].

The basic problem of *supervised binary classification* can be defined as follows. Let us assume that we are given a labeled database in the form of pairs of feature vectors and their correct classification, i.e. $z_1 = (x_1, y_1), \dots, z_n = (x_n, y_n)$, where $x_i \in \mathbb{R}^d$, and $y_i \in \{-1, 1\}$. The constant d is the *dimension* of the problem (the number of features). We are looking for a *model* $f_w : \mathbb{R}^d \to \{-1, 1\}$ parameterized by a vector w that correctly classifies the available feature vectors, and that can also *generalize* well; that is, which can classify unseen examples too.

Supervised learning can be thought of as an optimization problem, where we want to minimize the empirical risk

$$E_n(w) = \frac{1}{n} \sum_{i=1}^{n} Q(z_i, w) = \frac{1}{n} \sum_{i=1}^{n} \ell(f_w(x_i), y_i) \tag{1}$$

where function $Q(z_i, w) = \ell(f_w(x_i), y_i)$ is a loss function capturing the prediction error on example z_i.

Training algorithms that iterate over available training data, or process a continuous stream of data records, and evolve a model by updating it for each individual data record according to some update rule are called *online learning algorithms*. Gossip learning relies on this type of learning algorithms. Ma et al. provide a nice summary of online learning for large scale data [15].

Stochastic gradient search [5, 6] is a generic algorithmic family for implementing online learning methods. The basic idea is that we iterate over the training examples in a random order repeatedly, and for each training example z_t we calculate the gradient of the error function (which describes classification error), and modify the model along this gradient to reduce the error on this particular example according to the rule

$$w_{t+1} = w_t - \gamma_t \nabla_w Q(z_t, w_t) \tag{2}$$

where γ_t is the learning rate at step t that often decreases as t increases.

A popular way to accelerate the convergence is the use of mini-batches, that is, to update the model with the gradient of the sum of the loss functions of a few training examples (instead of only one) in each iteration. This allows for fast distributed implementations as well [11].

In gossip learning, models perform random walks on the network and are trained on the local data using stochastic gradient descent. Besides, several models can perform random walks at the same time, and these models can be combined time-to-time to accelerate convergence. Our approach here will be based on this scheme, replacing the local update step with a mini-batch approach.

5   Our Solution

Based on the assumptions in Section 3 and building on the GOLF framework outlined in Section 4 we now present our algorithm for computing a mini-batch gradient in a single step of the mini-batch gradient descent algorithm. First of all, recall that models perform random walks over the nodes in the network. At each step, when a node receives a model to update, it will first create a mini-batch group by building a rooted tree. According to our assumptions adversaries cannot manipulate the neighborhood and they do not corrupt the protocol execution, so this can be achieved via simple local flooding algorithms.

Let us now describe what kind of tree is needed exactly. The basic version of our algorithm will require a *trunked tree*.

Definition 1 (trunked tree). *Any rooted tree is* 1-trunked. *For $k > 1$, a rooted tree is k-trunked if the root has exactly one child node, and the corresponding subtree is a $(k-1)$-trunked tree.*

Let N denote the intended size of the mini-batch group. We assume that N is significantly less than the network size. Let S be a parameter that determines the desired security level ($N \geq S \geq 2$). We can now state that we require an *S-trunked tree* rooted at the node that is being visited by gossip learning.

This tree can be constructed on an overlay network by taking $S - 1$ random steps, and then performing a flooding algorithm with appropriately set time-to-live and branching parameters. The exact algorithm for this is not very interesting, mostly because it can be very simple. The reason is that when building the tree, no attention needs to be paid to reliability. We generate the tree quickly and use it only once quickly. Normally, some subtrees will be lost in the process but our algorithm is designed to tolerate this.

The effect of certain parameters, such as the branching factor and node failures, will be discussed later in the evaluation. In rare cases, when the neighborhood size is too small or when there are many cycles in the network, it could be hard to achieve the desired branching factor, which can result in a deeper tree than desired resulting in an increased time-complexity. Apart from this performance issue, the algorithm will function correctly even in these cases. From now on, for simplicity, we assume that the desired branching factor can be achieved.

The sum we want to calculate is over vectors of real numbers. We discuss the one-dimensional gradient from now on for simplicity. Homomorphic encryption works over

integers, to be precise, over the set of residue classes $\mathbb{Z}_n$ for some large n. For this reason we need to discretize the real interval that includes all possible sums we might calculate, and we need to map the resulting discrete intervals to residue classes in $\mathbb{Z}_M$ where M defines the granularity of the resolution of the discretization. This mapping is natural, we do not go into details here. Since the gradient of the loss function for most learning algorithms is bounded, this is not a practical limitation.

The basic idea of the algorithm is to divide the local value into S shares, encrypt these with asymmetric additively homomorphic encryption (such as the Paillier cryptosystem), and send them to the root via the chain of ancestors. Although the shares travel together, they are encrypted with the public keys of different ancestors. Along the route, the arrays of shares are aggregated, and periodically re-encrypted. Finally, the root calculates the sum.

The algorithm consists of three procedures, shown in Algorithm 1. These are run locally on the individual nodes. Procedure INIT is called once after the node becomes part of the tree. Procedure ONMESSAGERECEIVED is called whenever a message is received by the node. A message contains an array of dimension S that contains shares encoded for S ancestors. The first element msg[1] is encrypted for the current node, so it can decrypt it. The rest of the shares are shifted down by one position and added (with homomorphic encryption) to the local array of shares to be sent. After all the messages have been processed, the ith element ($1 \le i \le S - 1$) of the array SHARES is now encrypted with the public key of the ith ancestor of the current node and contains a share of the sum of the subtree except the local value of the current node. The Sth element is stored unencrypted in variable KNOWNSHARE.

Procedure ONNOMOREMESSAGESEXPECTED is called when the node has received a message from all of its children, or when the remaining children are considered to be dead by a failure detector. The timeout used here has to take into account the depth of the given subtree and the maximal delay of a message. In the case of leaf nodes, this procedure is called right after INIT.

The function call ANCESTOR(i) returns the descriptor of the ith ancestor of the current node that contains the necessary public keys as well. During tree building this information can be given to each node. For the purposes of this function, the parent of the root is defined to be itself. Function ENCRYPT(x, y) encrypts the integer x with the public key of node y using an asymmetric additively homomorphic cryptosystem. DECRYPT(x) decrypts x with the private key of the current node. Operation $a \oplus b$ performs the homomorphic addition of the two encrypted integers a and b to get the encrypted form of the sum of these integers. Function RANDOM(x) returns a uniformly distributed random integer in the range $[0, x - 1]$.

If the current node is the root, then the elements of the received array are decrypted and summed. The root can decrypt all the elements because it is the parent of itself, so all the elements are encrypted for the root when the message reaches it. If the current node is not the root then the local value has to be added, and the Sth element of the array has to be filled. First, the local value is split into S shares according to the S-out-of-S secret-sharing scheme discussed in [16]: $S - 1$ out of the S shares are uniformly distributed random integers between 0 and $M - 1$. The last share is the difference between the local value and the sum of the random numbers (mod M). This way, the sum of shares equals the local value (mod M). Also, the sum of any non-empty proper

Algorithm 1

procedure INIT
    shares ← new array[1..S]
    for $i \leftarrow 1$ **to** S **do**
        shares[i] ← Encrypt(0, Ancestor(i))
    end for
    knownShare ← 0
end procedure

procedure ONMESSAGERECEIVED(msg)
    for $i \leftarrow 1$ **to** $S - 1$ **do**
        shares[i] ← shares[i] $\oplus$ msg[$i + 1$]
    end for
    knownShare ← knownShare + Decrypt(msg[1])
end procedure

procedure ONNOMOREMESSAGESEXPECTED
    if IAmTheRoot() **then**
        for $i \leftarrow 1$ **to** $S - 1$ **do**
            knownShare ← knownShare + Decrypt(shares[i])
        end for
        Publish((knownShare + localValue) $\mod M$)
    else
        randSum ← 0
        for $i \leftarrow 1$ **to** $S - 1$ **do**
            rand ← Random(M)
            randSum ← randSum + rand
            shares[i] ← shares[i] $\oplus$ Encrypt(rand, Ancestor(i))
        end for
        knownShare ← knownShare + localValue − randSum
        shares[S] ← Encrypt(knownShare $\mod M$, Ancestor(S))
        SendToParent(shares)
    end if
end procedure

subset of these shares is uniformly distributed, therefore nothing can be learned about the local value without knowing all the shares.

The shares calculated this way can be encrypted and added to the corresponding shares, and finally the remaining Sth share is re-encrypted with the public key of the Sth ancestor and put into the end of the array. When this array is sent to the parent, it contains the S shares of the partial sum corresponding to the full sub-tree.

We note here that if during the algorithm a child node never responds, then its subtree will be essentially missing (will have a sum of zero) but other than that the algorithm will terminate normally. This is acceptable in our application, because for a mini-batch we simply need the sum of any number of gradients, this will not threaten the convergence of the gradient descent algorithm.

6   Discussion

6.1   Security

To steal information, that is, to learn the sum over a subtree, the adversary needs to catch and decrypt all the S shares of the corresponding message that was sent by the root of the subtree in question. Recall that if the adversary decrypts less than S shares from any message, it still has only a uniform random value due to our construction. To be more precise, to completely decrypt a message sent to node c_1, the adversary needs to corrupt c_1 and all its $S - 1$ closest ancestors, denoted by $c_2, .., c_S$, so he can obtain the necessary private keys.

The only situation when the shares of a message are not encrypted with the public keys of S *different* nodes—and hence when less than S nodes are sufficient to be corrupted—is when the distance of the sender from the root is less than S. In this case, the sender node is located in the trunk of the tree. However, decrypting such a message does not yield any more information than what can be calculated from the (public) result of the protocol and the local values (gradients) of the nodes needed to be corrupted for the decryption. This is because in the trunk the sender of the message in question is surely the only child of the first corrupted node, and the message represents the sum of the local values of all the nodes, except for the ones needed to be corrupted. To put it in a different way, corrupting less than S nodes never gives more leverage than learning the private data of the corrupted nodes only.

Therefore, the only way to steal extra information (other than the local values of the corrupted nodes) is to form a continuous chain of corrupted nodes $c_1, .., c_S$ towards the root, where c_{i+1} is the parent of c_i. This makes it possible to steal the partial sums of the subtrees rooted at the children of c_1. For this reason we now focus only on the $N - S$ vulnerable subtrees not rooted in the trunk.

As a consequence, a threshold adversary cannot steal information if he corrupts at most $S - 1$ nodes. A probabilistic adversary that corrupts each node with probability p can steal the exact partial sum of a given subtree whose root is not corrupted with probability p^S.

Even if the sum of a given subtree is not stolen, some information can be learned about it by stealing the sums of other subtrees. However, this information is limited, as demonstrated by the following theorem.

Theorem 1. *The private value of a node that is not corrupted cannot be exactly determined by the adversary as long as at least one of the S closest ancestors of the node is not corrupted.*

Proof. Let us denote by t the target node, and by u the closest ancestor of t that is not corrupted. The message sent by t cannot be decrypted by the adversary, because one of its shares is encrypted to u (because u is one of the S closest ancestors of t). The same holds for all the nodes between t and u. Therefore the smallest subtree that contains t and whose sum can be stolen also contains u. Due to the nested nature of subtrees, bigger subtrees that contains t also contains u as well. Also, any subtree that contains u also contains t (since t is the descendant of u). Therefore u and t cannot be separated. Even if every other node is corrupted in the subtree whose sum is stolen, only the sum of the private values of u and t can be determined.

Therefore p^S is also an upper bound on the probability of stealing the exact private value of a given node that is not corrupted.

6.2   Complexity

In a tree with a maximal branching factor of B each node sends only one message, and receives at most B. The length of a message is $\mathcal{O}(SC)$, an array of S encrypted integers, where C is the length of the encrypted form of an integer. Let us now elaborate on C. First, as stated before, the sum of the gradients is represented on $\mathcal{O}(\log M)$ bits, where M is a design choice defining the precision of the fixed point representation of the real gradient. Let us assume for now that we use the Paillier cryptosystem [19]. In this case, we need to set the parameters of our cryptosystem in such a way that the largest number it can represent is no less than $n = \min(B^S M, NM)$, which is the upper bound of any share being computed by the algorithm (assuming $B \geq 2$). In the Paillier cryptosystem the ciphertext for this parameter setting has an upper bound of $\mathcal{O}(n^2)$ for a single share. Since

$$S \log n^2 = S \log \min(B^S M, NM)^2 \leq 2(S^2 \log B + S \log M), \qquad (3)$$

the number of bits required is $\mathcal{O}(S^2 \log B + S \log M)$.

The computational complexity is $\mathcal{O}(BSE)$ per node, where E is the cost of encryption, decryption, or homomorphic addition. All these three operations boil down to one or two exponentiations in modular arithmetic in the Paillier cryptosystem. Note that this is independent of N.

The time complexity of the protocol is proportional to the depth of the tree. If the tree is balanced, this results in $S + \mathcal{O}(logN)$ steps altogether.

6.3   Robustness

As mentioned before, if a node failure occurs then the subtree rooted at that node is left out of the sum. In our application this does not render the output useless, since in mini-batch methods one can apply mini-batches of varying size.

Let us take a closer look at the possible effect of node failure. From the point of starting to build the tree until the root computes the end result a certain number of nodes might fail at random. The worst-case scenario is when all these nodes fail right after the construction of the tree but before starting to propagate shares upwards.

We have conducted experiments to assess the robustness of the trees under various parameter settings. In the initialization step, a random graph of 1,000,000 nodes is generated in the following way: 20% percent of the nodes are marked public and then each node gets 20 links to random public nodes. These links represent bidirectional communication channels. It has been argued that such a construction is a viable approach in the presence of NAT devices on the open Internet [21]. Also, recently Berta et al. [2] estimated that the NAT types of about 20% of smartphones are either open access or full cone. Thus, the parameter setting and the overlay above is a good representation of one application domain: smartphone networks.

After this, random trees are generated with a depth of D and a maximal branching factor of B, in the following way: a root is chosen randomly, which selects B of its
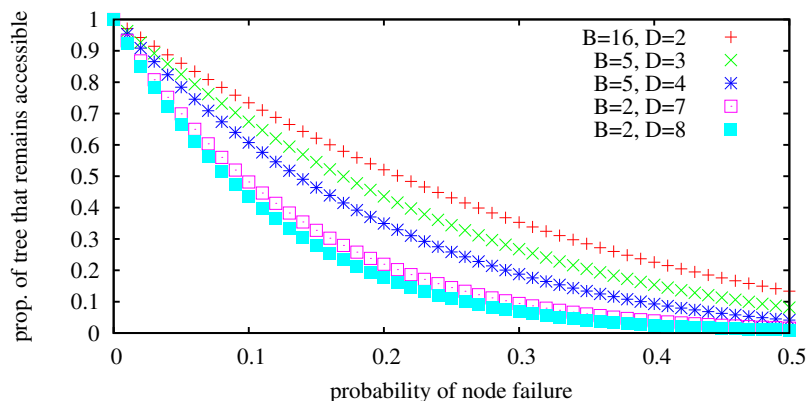
Fig. 1. The expected value of the ratio of nodes that successfully contribute to the computation, plotted as a function of the probability of node failure. B denotes maximal branching factor and D denotes depth. (An isolated node has depth 0.)

neighbors as children, then each of them, in turn, selects B of their respective neighbors, and so on, until depth D is reached. No node selects its parent, but multiple nodes may try to select the same node, in which case it becomes the child of only one of them. Therefore nodes can have less than B children, but this happens infrequently, if the graph is large enough compared to the tree. These trees are used to calculate the expected value of the ratio of the nodes that are reachable from the root via a chain of available nodes, assuming a given chance for node failure, in the worst case scenario we outlined above. If the probability of node failure is f, a node located at level d of the tree (the root has level 0) will successfully contribute its local value to the sum with probability $(1 - f)^{(d+1)}$.) The results are shown in Figure 1. Each curve represents a given setting of B and D. Each point is based on 50 different random trees.

To provide an indication of feasible failure rates in an actual network, we analyzed the trace collected by Berta et al. [2]. In this trace a node was defined to be available when it had network connectivity and when it was on a charger at the same time. Figure 2 shows statistics about smartphone availability. For each hour, we calculated the probability that a node that has been online for at least a minute remains online for 1, 5 or 10 more minutes. As the figure illustrates, these probabilities are rather high even for a 10 minute extra time, which is certainly sufficient to complete a mini-batch for any reasonable batch size, given that the time complexity is logarithmic in size. Comparing this with Figure 1, under these realistic failure rates the resulting computation will cover a large subset of the intended mini-batch.

7   Variations

Although the robustness of the algorithm is useful, we have to be careful when publishing a sum based on too few participants. The algorithm can be modified to address this issue. Let us denote by R the minimal required number of actual participants
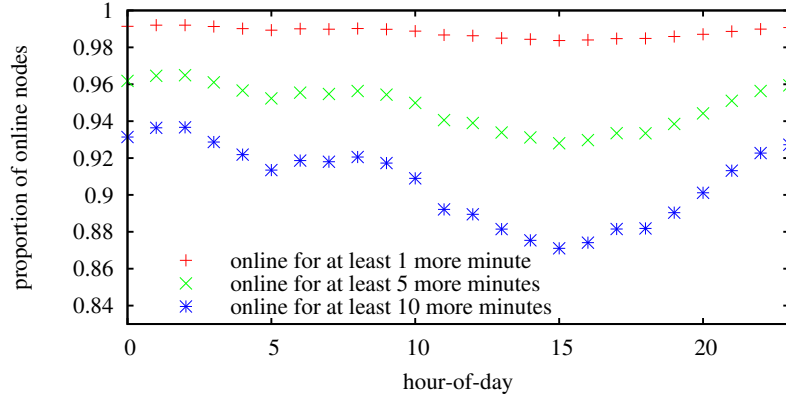
Fig. 2. Expected availability of smartphones that have been online for at least a minute. Hour-of-day is in UTC.

($S \leq R \leq N$). Each message is padded with an (unencrypted) integer n indicating the number of nodes its data is based on. When the node exactly $S - 1$ steps away from the root (thus in the trunk) is about to send its message, it checks whether $n + S - 1 \geq R$ holds (since the remaining nodes towards the root have no children except the one on this path). If not, it sends a failure message instead. The nodes fewer than $S - 1$ steps away from the root transmit a failure message if they receive one, or if they fail to receive any messages.

One can ask the question whether the trunk is needed, as the protocol can be executed on any tree unmodified. However, having no trunk makes it easier to steal information about subtrees close to the root. If the tree is well-balanced and the probability of failure is small, these subtrees can be large enough for the stolen partial sums to not pose a practical privacy problem in certain applications. The advantages include a simpler topology, a faster running time, and increased robustness.

Another option is to replace the top $S - 1$ nodes with a central server. To be more precise, we can have a server simulate the top $S - 1$ nodes with the local values of these nodes set to zero. This server acts as the root of a 2-trunked tree. From a security point of view, if the server is corrupted by a semi-honest adversary, we have the same situation when the top $S - 1$ nodes are corrupted by the same adversary. As we have shown in Section 6.1, one needs to corrupt at least S nodes in a chain to gain any extra advantage, so on its own the server is not able to obtain extra information other than the global sum. Also, the server does not need more computational capacity or bandwidth than the other nodes. This variation can be combined with the size propagation technique described above. Here, the child of the server can check whether $n \geq R$ holds.

8   Evaluation of Convergence Speed

Here, we illustrate the cost of using mini-batch learning instead of stochastic gradient descent, and we also illustrate the overhead of our cryptographic techniques on the mini-batch algorithm.

We simulated our algorithm over the Spambase binary classification data set from the UCI repository[14], which consists of 4601 records, 39.4% of which are positive. 10% of the records were reserved for testing. Each node had one record resulting in a network size of 4601. The trees we tested had a trunk length of S with D additional levels below the trunk with a branching factor of B. Each node stays alive during the calculation of the batch-sum with probability P resulting in E nodes (E is a random variable) that end up participating in the computation (see Figure 1).

The learning method we used was logistic regression [4]. We used the L2-regularized logistic regression online update rule

$$w \leftarrow \frac{t}{t+1}w + \frac{\eta}{t+1}(y-p)x$$

where w is the weight vector of the model, t is the number of samples seen by the model (not including the new one), x is the feature vector of the training example, y is the correct label (1 or 0), p is the prediction of the model (probability of the label being 1), and η is the learning parameter. We generalize this rule to mini-batches of size E as follows:

$$w \leftarrow \frac{t}{t+E}w + \left(\frac{1}{E}\sum_{i=1}^{E}\frac{\eta}{t+i}\right)\sum_{i=1}^{E}(y_i-p_i)x_i$$

where $(y_i-p_i)x_i$ is supposed to be calculated by the individual nodes, and summed using Algorithm 1. After the update, t is increased by E instead of 1. η was set to 1000.

Our baseline is the case when one instance of stochastic gradient descent (SGD) is started by each node and the nodes immediately forward all received models after updating it, thereby utilizing all the available bandwidth (in practice users can set upper bounds on this utilization, we assumed the maximal bandwidth is the same at all the nodes). We run mini-batch with and without cryptography (secure mini-batch and mini-batch). The number of instances we start of these mini-batch variants are chosen so that they use the same bandwidth as SGD. With cryptography we use Algorithm 1 to compute the gradient sum. Without cryptography we use the same tree but we do not encode the messages. Instead, we propagate the plain partial sum instead of S different encoded shares. Note that mini-batch with and without cryptography is in fact identical except that with cryptography all the messages are at most about $2S$ times larger and thus they take this much longer to transmit (see Section 6.2).

Figure 3 shows our results. Clearly, mini-batch gradient does not result in serious performance loss in itself. Cryptography does add overhead that is linearly proportional to the parameter S, since the message size includes the factor of S due to sending this number of shares.
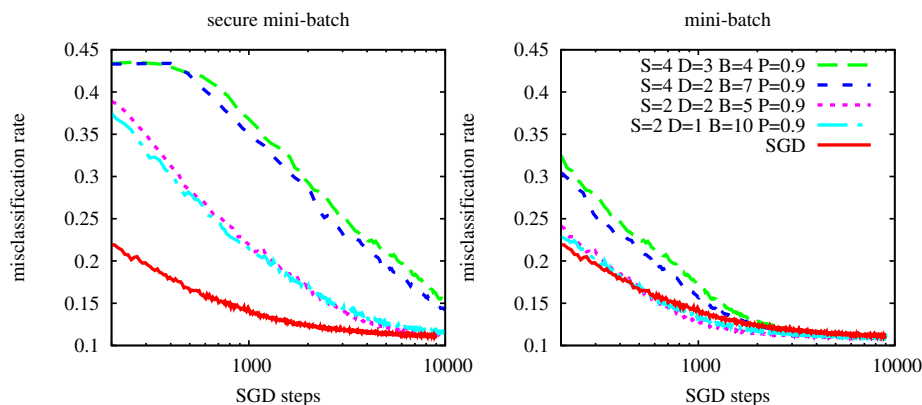
Fig. 3. Misclassification rate (zero-one error) of secure mini-batch (left) and mini-batch (right) averaged over the nodes and over 1000 different runs as a function of time (measured in SGD steps).

9   Conclusion

We proposed a secure sum protocol to prevent the collusion attack in gossip learning. The main idea is that instead of SGD we implement a mini-batch method and the sum within the mini-batch is calculated using our novel secure algorithm. We can achieve very high levels of robustness and very good scalability through exploiting the fact that the mini-batch gradient algorithm does not require the sum to be precise. The algorithm runs in logarithmic time and it is designed to calculate a partial sum in case of node failures. It can tolerate collusion unless there are S consecutive colluding nodes on any path to the root of the aggregation tree, where S is a free parameter. Under practical parameter settings the communication complexity of the secure mini-batch algorithm is only approximately a constant factor of $2S$ larger than that of the plain mini-batch algorithm.

References

1. Ahmad, W., Khokhar, A.: Secure aggregation in large scale overlay networks. In: IEEE Global Telecommunications Conference (GLOBECOM '06) (2006)
2. Berta, Á., Bilicki, V., Jelasity, M.: Defining and understanding smartphone churn over the internet: a measurement study. In: Proceedings of the 14th IEEE International Conference on Peer-to-Peer Computing (P2P 2014). IEEE (2014)
3. Bickson, D., Reinman, T., Dolev, D., Pinkas, B.: Peer-to-peer secure multi-party numerical computation facing malicious adversaries. Peer-to-Peer Networking and Applications 3(2), 129–144 (2010)
4. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
5. Bottou, L.: Stochastic gradient descent tricks. In: Montavon, G., Orr, G.B., Müller, K.R. (eds.) Neural Networks: Tricks of the Trade, Lecture Notes in Computer Science, vol. 7700, pp. 421–436. Springer Berlin Heidelberg (2012)

6. Bottou, L., LeCun, Y.: Large scale online learning. In: Thrun, S., Saul, L., Schölkopf, B. (eds.) Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, MA (2004)
7. Clifton, C., Kantarcioglu, M., Vaidya, J., Lin, X., Zhu, M.Y.: Tools for privacy preserving distributed data mining. SIGKDD Explor. Newsl. 4(2), 28–34 (December 2002)
8. Dekel, O., Gilad-Bachrach, R., Shamir, O., Xiao, L.: Optimal distributed online prediction using mini-batches. J. Mach. Learn. Res. 13(1), 165–202 (January 2012)
9. Dwork, C.: A firm foundation for private data analysis. Commun. ACM 54(1), 86–95 (January 2011)
10. Dwork, C., Kenthapadi, K., McSherry, F., Mironov, I., Naor, M.: Our data, ourselves: Privacy via distributed noise generation. In: Vaudenay, S. (ed.) Advances in Cryptology - EURO-CRYPT 2006, Lecture Notes in Computer Science, vol. 4004, pp. 486–503. Springer Berlin Heidelberg (2006)
11. Gimpel, K., Das, D., Smith, N.A.: Distributed asynchronous online learning for natural language processing. In: Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL'10). pp. 213–222. Association for Computational Linguistics, Stroudsburg, PA, USA (2010)
12. Han, S., Ng, W.K., Wan, L., Lee, V.C.S.: Privacy-preserving gradient-descent methods. IEEE Transactions on Knowledge and Data Engineering 22(6), 884–899 (2010)
13. Jesi, G.P., Montresor, A., van Steen, M.: Secure peer sampling. Computer Networks 54(12), 2086–2098 (2010)
14. Lichman, M.: UCI machine learning repository (2013), `http://archive.ics.uci.edu/ml`
15. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Identifying suspicious URLs: an application of large-scale online learning. In: Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09). pp. 681–688. ACM, New York, NY, USA (2009)
16. Maurer, U.: Secure multi-party computation made simple. Discrete Applied Mathematics 154(2), 370–381 (2006)
17. Naranjo, J.A.M., Casado, L.G., Jelasity, M.: Asynchronous privacy-preserving iterative computation on peer-to-peer networks. Computing 94(8–10), 763–782 (2012)
18. Ormándi, R., Hegedűs, I., Jelasity, M.: Gossip learning with linear models on fully distributed data. Concurrency and Computation: Practice and Experience 25(4), 556–571 (2013)
19. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Advances in Cryptology – EUROCRYPT '99. Lecture Notes in Computer Science, vol. 1592, pp. 223–238. Springer Berlin Heidelberg (1999)
20. Rajkumar, A., Agarwal, S.: A differentially private stochastic gradient descent algorithm for multiparty classification. JMLR Workshop and Conference Proceedings 22, 933–941 (2012), proceedings of AISTATS'12
21. Roverso, R., Dowling, J., Jelasity, M.: Through the wormhole: Low cost, fresh peer sampling for the internet. In: Proceedings of the 13th IEEE International Conference on Peer-to-Peer Computing (P2P 2013). IEEE (2013)
22. Saia, J., Zamani, M.: Recent results in scalable multi-party computation. In: 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'15). LNCS, vol. 8939. Springer (2015)
23. Stutzbach, D., Rejaie, R., Duffield, N., Sen, S., Willinger, W.: On unbiased sampling for unstructured peer-to-peer networks. IEEE/ACM Transactions on Networking 17(2), 377–390 (April 2009)
24. Yao, A.C.: Protocols for secure computations. In: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS). pp. 160–164 (1982)