Hindawi
Security and Communication Networks
Volume 2018, Article ID 6728020, 15 pages
https://doi.org/10.1155/2018/6728020

Research Article

Robust Fully Distributed Minibatch Gradient Descent with Privacy Preservation

Gábor Danner, Árpád Berta, István Hegedűs, and Márk Jelasity

University of Szeged, and MTA-SZTE Research Group on AI, Szeged, Hungary

Correspondence should be addressed to Márk Jelasity; jelasity@inf.u-szeged.hu

Received 3 November 2017; Revised 3 March 2018; Accepted 4 April 2018; Published 14 May 2018

Academic Editor: Po-Ching Lin

Privacy and security are among the highest priorities in data mining approaches over data collected from mobile devices. Fully distributed machine learning is a promising direction in this context. However, it is a hard problem to design protocols that are efficient yet provide sufficient levels of privacy and security. In fully distributed environments, secure multiparty computation (MPC) is often applied to solve these problems. However, in our dynamic and unreliable 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 query over a subset of participants assuming a semihonest 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 minibatch 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. We utilize the Paillier homomorphic cryptosystem as part of our solution combined with extreme lossy gradient compression to make the cost of the cryptographic algorithms affordable. We demonstrate both theoretically and experimentally, based on churn statistics from a real smartphone trace, that the protocol is indeed practically viable.

1. Introduction

Data mining over personal data harvested from mobile devices is a very sensitive problem due to the strong requirements of privacy preservation and security. Recently, the *federated learning* approach was proposed to solve this problem by not collecting the data in the first place but instead processing the data in place and creating the final models in the cloud based on the models created locally [1, 2].

We go one step further and propose a solution that does not utilize centralized resources at all. The main motivation for a fully distributed solution in our cloud-based era is to preserve privacy by avoiding the central collection of any personal data, even in preprocessed form. 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. The key issue here of course is to offer decentralized algorithms that are competitive with approaches like federated learning in terms of time and communication complexity and that provide increased levels of privacy and security.

Previously, we proposed numerous 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 [3] (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, for example, [4], 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 semihonest) 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 minibatch learning, which, apart from increasing the resistance to collusion, is known to often speed up the learning algorithm as well (see, e.g., [5]).

It might seem attractive to run a secure multiparty computation (MPC) algorithm within the minibatch 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 [6]. 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, light-weight, and robust, since the participating nodes may go offline at any time [8] and they might have limited resources. One key observation is that for the minibatch 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 binomial tree topology and Paillier 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.

We also put a great emphasis on demonstrating that the proposed protocol is practically viable. This is a nontrivial question because homomorphic cryptosystems can quickly become very expensive when applied along with large-enough key-sizes (such as 2048 bit keys), especially considering that in machine learning the gradients can be rather large. To achieve practical viability, we propose an extreme lossy compression, where we discretize floating-point gradient values to as few as two bits. We demonstrate experimentally that this does not affect learning accuracy yet allows for an affordable cryptography cost. Our simulations are based on a real smartphone trace we collected [8].

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 [9]. 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 [10]. 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 minibatches 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 [11]. 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.

Bonawitz et al. [13] address a similar problem setting where the goal is to compute a secure sum in an efficient and robust manner. They also assume a semihonest adversarial model (with a limited set of potentially malicious behaviors by a server). However, their solution requires a server and an all-to-all broadcast primitive even in the most efficient version of their protocol. Our solution requires a linear number of messages only.

The algorithm of Ahmad and Khokhar is similar to ours [14], as they also use a tree to aggregate values using homomorphic encryption. However, 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 minibatch gradient descent application is that for each minibatch 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. In our solution, all the nodes have their own public/private key pair and no keys have to be shared at any point in time. Besides, these key pairs may remain the same in every minibatch the given node participates in without compromising our security guarantees.

We need to mention the area of differential privacy [15], which is concerned with 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 (e.g., [16]). In our distributed setup, this noise term can be computed in a distributed and secure way [17].

We also strongly build on our previous work [18]. There, we proposed an algorithm very similar to the one presented here. In this study we offer several optimizations of the algorithm and we propose the binomial topology for building the minibatch overlay tree. We also explore the issue of gradient compression necessary for keeping the cost of cryptography under control and we perform a thorough experimental study of the algorithm based on a smartphone churn trace.

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 (e.g., 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 [19].

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.

We assume that the adversaries are honest but curious (or semihonest). That is, nodes corrupted by an adversary will follow the protocol but the adversary can see the internal state of the node. 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). Wiretapping is allowed, since all the sensitive messages in our protocol are encrypted.

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) [3].

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, that is, $z_1 = (x_1, y_1), \ldots, z_n = (x_n, y_n)$, 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:

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

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 [21].

Stochastic gradient search [22, 23] 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 following rule:

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

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 minibatches, 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 [24].

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 minibatch approach.

5. Our Solution

As explained previously, at each step, when a node receives a model to update, it coordinates the distributed computation of a minibatch gradient and then uses this gradient to update the model. 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 minibatch gradient.

5.1. Minibatch Tree Topology. The very first step for computing a minibatch gradient is to create a temporary group of random nodes that form the minibatch. In our decentralized environment we do this by building a rooted overlay tree. The basic version of our algorithm will require the overlay tree not

only to be rooted at the node computing the gradient but also to be *trunked*.

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 minibatch 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. As we will see later, this is to prevent a malicious root to collect too much information.

Apart from the trunk, the tree can be arbitrary; however, we propose a *binomial tree* as a preferable choice. If every node already in the tree spawns a new child node in periodic rounds (starting from a single root node) then the result is a binomial tree. It is not possible to construct a tree of a given size faster, since in the case of a binomial tree each node keeps working continuously so the efficiency is maximal. Of course we assumed here that child nodes can be added only sequentially at a given node. However, if we also assume that all the nodes have the same up- and download bandwidth cap then adding nodes in parallel will be proportionally slower thus parallelism provides no advantage as long as we utilize the maximal available bandwidth. The same up- and download bandwidth requirement is naturally satisfied in our application domain because we assume that the protocol is allowed to use only a fixed, relatively small amount of bandwidth (such as 1 Mbps) and low bandwidth connections are excluded from the set of possible overlay connections.

Another advantage of binomial trees is that we can use the links in reverse order of construction for uploading and aggregating data along the tree. This way, we get a data aggregation schedule that is similarly efficient and also collision-free in the sense that each node communicates with at most one node at a given time.

The tree overlay network we have described so far can be constructed over a random overlay network by first building the trunk (which takes a random walk of $S-1$ steps) and then recursively constructing a binomial tree of depth D, resulting in an S-trunked tree of size 2^D+S-1 and total depth $d = D+S-1$. Every child node is chosen randomly from those neighbors of the node that are both online and not in the tree already. 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 because of churn but our algorithm is designed to tolerate this. The effect of certain parameters, such as the binomial tree parameter and node failures, will be discussed later in the evaluation.

5.2. Calculating the Gradient. The sum we want to calculate is over vectors of real numbers. Without loss of generality, we discuss the one-dimensional case 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. Also, in Section 7 we evaluate the effect of discretization on learning performance and we show that even an extreme compression (discretizing the gradient down to two bits) is tolerable due to the high robustness of the minibatch gradient method itself.

In a nutshell, the basic idea of the algorithm is to divide the local value at each node 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 reencrypted. 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. Here, the function call Ancestor(i) returns the descriptor of the ith ancestor on the path towards the root. The descriptor contains the necessary public keys as well. During tree building this information can be given to each node so the nodes can look up the keys of their ancestors locally. For the purposes of the Ancestor 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.

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 the S closest ancestors to the sender child. The first element (msg[1]) is thus 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 (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). Note that the ith element ($1 \leq i \leq S-1$) of the array shares is encrypted with the public key of the ith ancestor of the current node and is used to aggregate a share of the sum of the subtree except the local value of the current node. The Sth share is aggregated in variable knownShare unencrypted. The value of share[S] is not modified in this method; it will be initialized using knownShare after all the child nodes that are alive have responded.

After all the shares have been processed, procedure OnNoMoreMessagesExpected is called. This happens 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. When calling OnNoMoreMessagesExpected, we know that the ith element ($1 \leq i \leq S-1$) of the array shares already contains the ith share of the sum of the subtree rooted at the current

```
procedure Init
    shares ← new array[1 ··· S]
    for i ← 1 to S do
        shares[i] ← Encrypt(0, Ancestor(i))
    end for
    knownShare ← 0
end procedure
Procedure OnMessageReceived(msg)
    for i ← 1 to S − 1 do
        shares[i] ← shares[i] ⊕ msg[i + 1]
    end for
    knownShare ← knownShare + Decrypt(msg[1])
end procedure
procedure OnNoMoreMessagesExpected
    if IAmTheRoot() then
        for i ← 1 to S − 1 do
            knownShare ← knownShare + Decrypt(shares[i])
        end for
        Publish((knownShare + localValue) mod M)
    else
        randSum ← 0
        for i ← 1 to S − 1 do
            rand ← Random(M)
            randSum ← randSum + rand
            shares[i] ← shares[i] ⊕ Encrypt(rand, Ancestor(i))
        end for
        knownShare ← knownShare + localValue − randSum
        shares[S] ← Encrypt(knownShare mod M, Ancestor(S))
        SendToParent(shares)
    end if
end procedure
```

Algorithm 1

node (except the local value of the current) encrypted with the public key of the ith ancestor of the current node. We also know that knownShare contains the Sth share of the same sum unencrypted.

Now, 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. Here, Decrypt(x) decrypts x using the private key of the current node. Function Publish(x) announces x, the output of the algorithm, that is, the final unencrypted sum.

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 [20]: $S-1$ 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 nonempty proper subset of these shares is uniformly distributed; therefore nothing can be learned about the local value without knowing all the shares. Function Random(x) returns a uniformly distributed random integer in the range $[0, x-1]$.

The shares calculated this way are then encrypted and added to the corresponding shares, and finally the remaining Sth share is encrypted with the public key of the Sth ancestor and put into the end of the array. This array—that now contains the S shares of the sum of the full subtree including the current node—is sent to the parent.

5.3. Working with Vectors. We now describe how to efficiently extend our method to vectors of discrete numbers, by packaging multiple elements into a single block of encrypted data. Let us first calculate the number of bits that are required to represent one vector element. Assume that the elements of the input vectors are in the range $[0, m]$. This means that the elements of the output vector fall in range $[0, Nm]$, where N is the minibatch (tree) size. That is, $M = Nm+1$. After applying the secret-sharing scheme on an input vector, the elements of the resulting shares also fall in the range $[0, Nm]$ due to the S-out-of-S secret-sharing scheme we apply.

However, when working with homomorphic cryptography, we keep adding encrypted shares together without performing the modulo operation that is required for the correct decoding in our S-out-of-S secret-sharing scheme and for keeping the values in the range $[0, Nm]$. Thus, we need a larger range to accommodate the sum of at most N shares

giving us the range of $[0, N^2m]$. This means that $\lceil \log_2(1 + N^2m) \rceil$ bits are required per element.

Using this many bits, we can simply concatenate the elements of a share together to form a single bit vector before encryption. Homomorphic addition will result in the corresponding elements being added together. After decryption, the vector can be restored by splitting the bit vector, and element-wise modulo can be performed. This method can be trivially extended to arrays of blocks of a desired size, by packaging the elements into multiple blocks.

5.4. Practical Considerations and Optimizations. We stress again 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 minibatch we simply need the sum of any number of gradients, this will not threaten the convergence of the gradient descent algorithm.

The pseudocode discussed above describes a simple and basic version of our algorithm that allows for optimizations to speed up execution. Execution time is important because a shorter execution time allows less time for nodes to fail; in addition, the machine learning algorithm will execute faster as well. A simple optimization is, for example, if, as part of their initialization, all the nodes instantly start encrypting the $S-1$ shares of their local data with the public keys of its $S-1$ closest ancestors.

Another optimization is the parallelization of encryption and sending. Note that encrypting data typically takes much longer than sending it; we will evaluate this in more detail later on. Here, when calculating the message to send to the parent, the node immediately sends the first encoded share to the parent (i.e., the share that the parent can decrypt) so that the parent can start working on the decryption. The node then sends all the remaining shares except the Sth share, while calculating its own encryption of the Sth share. Finally, when the encryption is ready, the node sends the Sth share as well.

Also, consider that, due to the binomial tree structure, all the leaves are created at about the same time, so they will start to send their message to the parent at about the same time resulting in a more or less round-based aggregation protocol. This makes the time complexity of one such aggregation round in which the aggregation moves up one level (starting from the leaves) $E + T + L$, where E is the encryption/decryption time of a share, T is the transmission time of an encrypted share, and L is the network latency (assuming $E + T > ST$ and that the cost of homomorphic addition is negligible). Note that the actual algorithm does not rely on the existence of synchronized aggregation rounds; in fact, in realistic environments these rounds often overlap if, for example, a node finishes sooner due to losing its children. The rounds are merely an emergent property in reliable environments, a side-effect of using binomial trees as our tree topology.

Another possibility for optimization is based on the observation that shares that would be encrypted with the public keys of the ancestors of the root do not need to be encrypted at all, therefore the root in fact performs only a single decryption.

5.5. Variants. Apart from optimizations, one can consider slightly modified versions of the algorithm that can be useful for trading off security and robustness or that allow for a minimal involvement of a central server.

The first variation—that we will actually utilize during our evaluation in Section 8—is setting a lower bound on the size of the subtree that we accept. Indeed, we have to be careful when publishing a sum based on too few participants. Let us denote by R the minimal required number of actual participants ($S \leq R \leq N$). Let the nodes pad their messages 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. This way, no nodes can decode the sum of a set that is not large enough.

On a different issue, 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 semihonest 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.

6. Analysis

We first consider the level of security that our solution provides, and we also characterize the complexity of the algorithm.

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 $c_2, \ldots, c_S$, 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 $c_1, \ldots, c_S$ 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 2. *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 (which is an array of S encrypted integers) is $\mathcal{O}(SC)$, where C is the length of the encrypted form of an integer. Let us now elaborate on C. First, as stated before, the sum 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 values. Let us assume for now that we use the Paillier cryptosystem [25]. 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

$$\begin{aligned} S \log n^2 &= S \log \min\left(B^S M, NM\right)^2 \\ &\leq 2\left(S^2 \log B + S \log M\right), \end{aligned} \tag{3}$$

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 $S + \mathcal{O}(\log N)$ steps altogether.

7. Compressing the Gradient

As mentioned in Section 5.2, it is essential that we compress the gradient because in a realistic machine learning problem there are at least a few hundred parameters, often a lot more. Encoding and decoding this many floating-point numbers with full precision can be prohibitively expensive for our protocol, especially on a mobile device. For this reason, we evaluated the effect of gradient compression on the performance of gradient descent learning. Similar techniques have been used before in a slightly different context [1].

Let us first introduce the exact algorithms and learning tasks we used for this evaluation. As for the learning tasks, we used two data sets. The first is the Spambase binary classification data set from the UCI repository [26], which consists of 4601 records with 57 features. Each of these records belongs to an email that was classified either as spam or as a regular email. The features that represent a piece of email are based on, for example, word and character frequencies or the length of capital letter sequences within the email. 39.4% of the records are positive examples. 10% of the records were reserved for testing. Each node had one record resulting in a network size of 4140; the remaining part of the dataset (461 records) was used for testing. The second dataset we used was based on Reuters articles (http://download.joachims.org/svm_light/examples/example1.tar.gz) It contains 1000 positive and 1000 negative examples, with 600 additional examples used for testing. The examples have 9947 features. The dataset contains Reuters articles and the task is to decide

whether a given document is about "corporate acquisitions" or not. The documents are represented by word stam feature vectors, where each feature corresponds to the occurrence of a word. Hence, the representation is very high-dimensional and sparse (i.e., each vector contains mostly zeros).

We tested two machine learning algorithms. The first is logistic regression [27]. We used the L2-regularized logistic regression online update rule:

$$w \longleftarrow \frac{t}{t+1} w + \frac{\eta}{(t+1)} (p - y) x, \tag{4}$$

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 minibatches of size E as follows:

$$w \longleftarrow \frac{t}{t+E} w + \frac{\eta}{t+E} \sum_{i=1}^{E} (p_i - y_i) x_i, \tag{5}$$

where $(p_i - y_i)x_i$ 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 10^5. The second algorithm was linear SVM [28]. The setup is very similar to that of logistic regression; only the batch update rule we used is

$$w \longleftarrow \frac{t}{t+E} w + \frac{\eta}{t+E} \sum_{i=1}^{E} \left[y_i w^T x_i < 1 \right] y_i x_i, \tag{6}$$

where $[\cdot]$ is the Iverson bracket notation (1 if its parameter is true, otherwise 0). Here y is the correct label as before, however, now $y \in \{-1, +1\}$.

The compression method we used was the following. All the individual gradients within the minibatch were computed using a 32-bit floating-point representation. These gradients were then quantized by mapping each attribute to one of only three possible values: 1, 0, and –1. This mapping was achieved by stochastic quantization. The quantized value requires only 2 bits to encode, a dramatic compression compared to the original floating-point representation of 32 bits. In fact, since we have only three levels, theoretically only a trit is needed for the encoding. We exploit this fact when summing the gradients: the upper bound of the sum of trits (represented on two bits) is lower than the sum of two-bit values. These compressed gradients were then used in (5) and (6) where no further compression is applied.

We ran experiments with all the four possible combinations of learning algorithms and datasets, using four different batch sizes: $E = 1$, 10, 50, and 100. The results are shown in Figure 1. The figure shows how the classification accuracy evolves as a function of the number of training examples seen. Accuracy is the proportion of correctly classified instances, that is, the sum of the number of the true positive and the true negative test examples divided by the size of the test set. The databases are well-balanced with respect to the class labels, making this metric adequate. The compressed versions are indicated by the "C-" prefix. It is clear that in these experiments there is virtually no difference between the compressed and original versions. This result is quite striking and is probably explained by the fact that minibatch gradients still contain a lot of noise compared to the full gradient even if they are computed exactly.

In the following, we assume that gradient attributes can be safely encoded in two bits only.

8. Experimental Evaluation

In this section, our goal is to demonstrate that the decentralized secure minibatch gradient search we proposed is practically viable; that is, the running time in a real system with realistic parameters is acceptable and that the learning algorithm offers good performance under realistic failure conditions.

Recall that the solution we proposed consists of three components. The first is the overlay tree building algorithm, which defines the minibatches. The second is the secure sum computation algorithm, which assumes that an overlay tree is given. The third is the applied machine learning algorithm. These three components are modular; different solutions for any of these components can be combined.

We exploit this modularity in our experimental evaluation. First, for each scenario we determine the time that is needed to encrypt and decrypt the messages defined by our secure sum protocol based on the Paillier cryptosystem. We then plug these values into a simulation of the tree building and aggregation protocols under realistic network and failure conditions. The end result of this simulation is a series of minibatch sizes that are defined by the effective tree sizes we observe, along with a time-stamp for each minibatch that depends on the simulated duration of the secure minibatch gradient computation. Finally, we use these series of minibatch sizes as well as their timing to assess the performance of the machine learning algorithm in our system. This is possible, because the only important factor for machine learning is the effective size of the tree in each step. We assume that each tree defines a uniform random subset, which is a good approximation if the underlying overlay network is random.

To model the network required for simulating the tree building protocol, we used a real trace of smartphone user behavior [8]. The rest of the parameters defining the computational cost and network utilization were set based on realistic examples. We used PeerSim [29] for our simulations. Let us first describe our smartphone trace.

8.1. Trace Properties. The trace we used was collected by an openly available smartphone app called STUNner, as described previously [8]. In a nutshell, the app monitors and collects information about charging status, battery level, bandwidth, and NAT type.

We have traces of varying lengths harvested from 1,191 different users. We divided these traces into one-day segments, resulting in 41,849 segments altogether. With the help of these segments, we are able to simulate a virtual period of up to one day by assigning a different, randomly selected segment

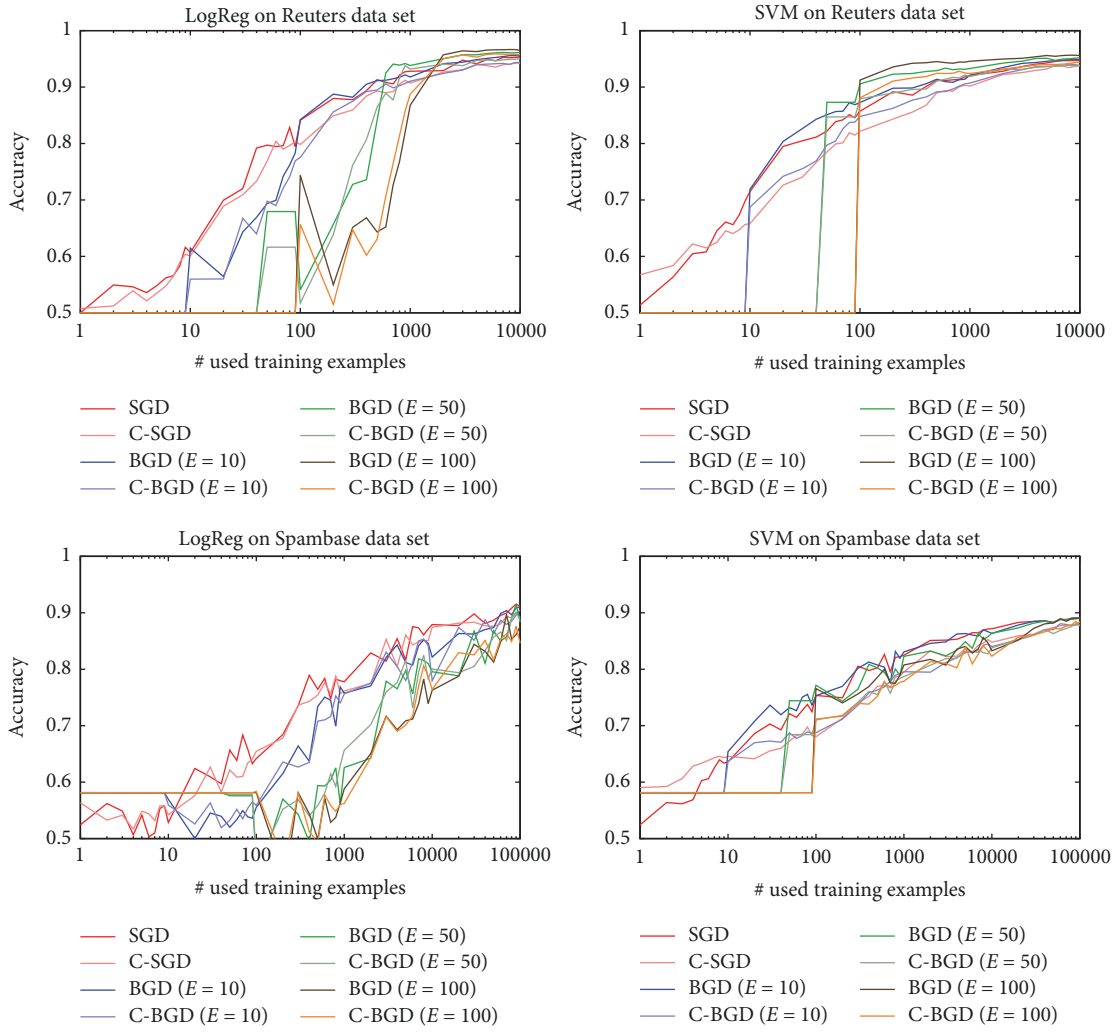

Figure 1: Classification accuracy of the compressed gradient update on the data sets with various batch sizes.

to each simulated node. The sampling of one-day segments is done without replacement. When the pool of segments runs out (which happens when we need more nodes than there are segments), we reinitialize the pool with the original 41,849 segments and continue the sampling without replacement. This way, we can simulate networks larger than 41,849 nodes. For example, as we will see, here we will simulate a network of size 100,000 for a one-day period.

Note that, due to this sampling method, users are represented with a probability proportional to the number of days they were online. This is motivated by the observation that the protocol at any given point in time can operate only with users that are actually online; hence those types of users that spend more time online are indeed encountered proportionally to their online-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 10 seconds remains online for 1, 5, or 10 more minutes. Note that for us these probabilities are important because the overlay tree that we build for each minibatch has to remain connected at least for the short amount of time that it takes to propagate the gradient updates to the root. As the figure illustrates, these probabilities are rather high even for a 10-minute extra time. In Section 8.3 we evaluate tree building experimentally in many scenarios

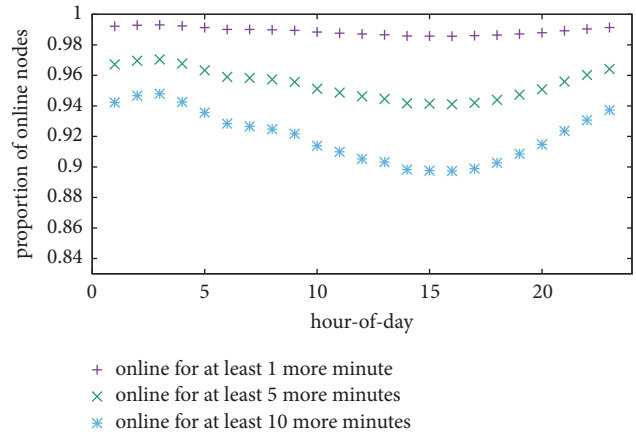

Figure 2: Expected availability of smartphones that have been online for at least 10 seconds. Hour-of-day is in UTC.

and show that, indeed, most of the overlay trees survive the short period during which they are used.

Although our sample contains users from all over the world, they are mostly from Europe, and some are from the USA. The indicated time is GMT; thus we did not convert times to local times.

Users with a bandwidth of less than 1 Mbps were treated as offline. This choice is motivated by two factors. First, the Internet bandwidth available to users has surpassed 1 Mbps in many developed countries, even for upload [30]. Indeed, in our trace the probability of encountering a connection with a bandwidth of lower than 1 Mbps is only 3.86%. Thus, excluding such devices will cause only minimal loss of data but in return slow devices will not slow the entire network down. Second, utilizing a device with such a low bandwidth would place too much of a burden on the device and user-friendly applications might want to avoid this. Applications based on our algorithm will mostly run in the background while collecting data and communicating with other devices. To ensure that an application of this kind is user-friendly, all the background processing needs to be transparent to the user.

Due to this consideration, in our experiments we will use 1 Mbps not only as a lower bound, but also as an upper bound. That is, the algorithm is allowed to utilize at most 1 Mbps of the available bandwidth, irrespective of the total available bandwidth, in order to avoid overloading the device. Obviously, utilizing all the available bandwidth would result in a more favorable convergence speed.

Note that we can simulate the case where a participating phone is required to have at least a certain battery level. From the point of view of churn, though, the worst case is when any battery levels are allowed to join, because this results in a more dynamic scenario. However, the first 10 seconds of each online session (or the entire session if it is shorter) is considered offline because extremely short online sessions would introduce unreliability. This technique can also be explicitly implemented as part of our protocol: a node should simply wait 10 seconds before joining the network.

8.2. Time Consumption. As mentioned above, we first describe the time consumption of the most important operations in our protocol. In order to do that, we carefully have to consider the size of each message that is transmitted and the time needed for encrypting and decrypting these messages. We performed these calculations in a number of scenarios with different parameters that represent interesting use cases.

The different scenarios as well as the corresponding message sizes and the amount of time needed to complete a number of different tasks are shown in Table 1. In the following we explain these scenarios and the computed values within these scenarios in detail.

For all the trees that we would like to build we fix $S = 4$, as indicated in the first column. This is our security parameter, introduced in Section 5.1. The value of $S = 4$ represents a good tradeoff between efficiency and the offered level of security. The binomial tree parameter D (the number of rounds used to build the tree) was set to 4 or 6, giving us the maximum tree sizes of 19 and 67, computed by the formula $N = 2^D + S - 1$, which was explained in detail in Section 5.1. The motivation for these settings is that our preliminary experiments with our machine learning application indicated that increasing the minibatch size beyond 67 is not beneficial. The lower value of 19 is motivated by the fact that smaller trees do not offer a sufficient level of privacy, since the sum is computed based on too few nodes. Also, in a very small tree, the trunk represents a considerable proportion of the tree which limits the possibilities for parallelization; hence the efficiency is not ideal.

The number of features in the learning problem was modeled to be 100 or 10,000. This setting accommodates the number of features in our datasets that are 57 for the Spambase dataset and 9947 for the Reuters dataset (see Section 7). Note that we rounded the number up to the closest power of 10 so that we have a 100 times scaling factor, which makes comparison more intuitive.

Based on the tree size N and the quantization parameter m we can compute the number of bits (b) needed to represent a share of one element of the secret-shared gradient vector. As explained in Section 5.3 in detail, the formula is given by $b = \lceil \log_2(1 + N^2 m) \rceil$. We used $m = 2$ based on our results on compressing the gradient vector in Section 7. The next column shows the key size (or block size) n, a parameter for the Paillier cryptosystem that defines the level of security. We examine the common values 1024 and 2048. Note that 2048 is currently recommended for sufficient security (https://www.keylength.com/).

Based on the parameters we already defined, we can now compute the number of blocks to be encoded per gradient share: $\lceil fb/n \rceil$. Finally, let us compute the message size to be sent by a node in the tree to its parent. According to the protocol, this message is composed of the S encrypted shares of the compressed gradient. The size of the message is $S2n\lceil fb/n \rceil$ bits. This is due to the fact that the size of an encrypted block is $2n$, and we need $\lceil fb/n \rceil$ blocks per share.

We have now computed almost all the values necessary to determine the time consumption of some important operations of the protocol. The last bit of information required for that is the time consumption of encoding a single block. The Paillier encryption and decryption time of a block is experimentally measured using an unoptimized Java implementation based on BigIntegers on a real Android device (Samsung SM-T280). This can be considered a worst case scenario because the implementation we used has a lot of room for optimization and the device itself is not an up-to-date model. Both the encryption and decryption take 0.041 s with a 1024-bit key and 0.300 s with a 2048-bit key.

Sending the model in plaintext from the parent to the child is required when building the tree. We assume single precision floating-point arithmetic (32 bits) so the sizes of the linear models are 3,200 bits and 320,000 bits for 100 and 10,000 features, respectively. The actual sending time is given by the 1 Mbps bandwidth we allow between online nodes and assuming a 100 ms latency. After receiving the model in plaintext the node instantly starts encrypting $S - 1$ shares as discussed in Section 5.4. This takes $S - 1$ times the encryption time of all the required blocks. The computed values are shown in Table 1.

Table 1

	Parameter setups							Time consumption (seconds)					Results
S	Number of features (f)	D	Max tree size (N)	Bits per feature (b)	Key size (n)	Blocks $\lceil fb/n \rceil$	Message size to parent $S2n\lceil fb/n \rceil$	Encrypt/decrypt a block	Send plain-text model	Encrypt $S-1$ shares	One aggregation round	Overall time of mini-batch	Prob. of good tree
4	10^2	4	19	10	1024	1	8192	0.041	0.103	0.123	0.143	1.847	0.999
					2048	1	16384	0.300	0.103	0.900	0.404	4.451	0.997
		6	67	14	1024	2	16384	0.041	0.103	0.246	0.186	2.850	0.997
					2048	1	16384	0.300	0.103	0.900	0.404	5.466	0.996
	10^4	4	19	10	1024	99	811008	0.041	0.420	12.177	4.362	45.649	0.969
					2048	50	819200	0.300	0.420	45.000	15.305	155.074	0.904
		6	67	14	1024	137	1122304	0.041	0.420	16.851	5.998	74.609	0.951
					2048	69	1130496	0.300	0.420	62.100	21.083	255.624	0.850

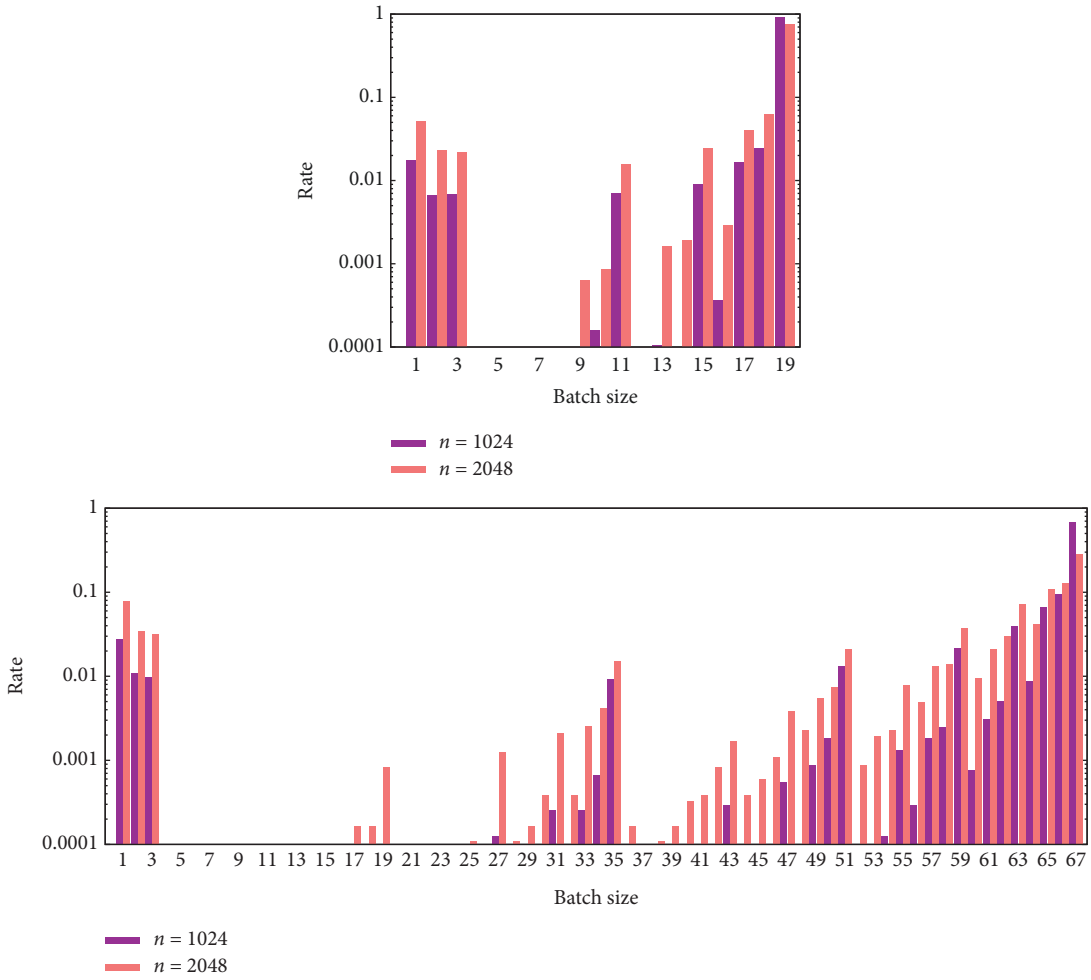

Figure 3: Distribution of effective minibatch sizes for scenario of 10,000 features. The histograms use a logarithmic scale.

The next column shows the time of one aggregation round, that is, the time needed for a child node to propagate information up to the parent. In Section 5.4 we described a number of variants of the protocol that involve different optimizations compared to the basic variant. Here, we assume the variant, in which children in the tree start encrypting their share while they simultaneously upload the other $S - 1$ shares to their parents. In all our scenarios uploading $S - 1$ shares is faster than encrypting one share. This means that the time needed for one aggregation round is the time of encoding one share plus the time of uploading this share (which consists of transmission time and network latency). The column indicating the time needed for one aggregation round shows this value for each parameter setting.

The column that corresponds to the overall minibatch time sums up all the required times for completing the minibatch, assuming the network is error free. This involves sending the plaintext model to the children down the tree during tree building as well as the aggregation rounds up to the root. These operations are performed for each level of the tree; note that the depth of the whole tree is $D + S - 1$. The time of encoding $S - 1$ shares also needs to be added because the leaves must first complete this encoding before starting the first aggregation round. If nodes can fail, in an actual run these times may be slightly longer because of the delay introduced by the failure detector, but they may also be slightly shorter, due to a smaller tree. Our simulations account for these effects. Note that we ignored the time consumption of the single gradient update step that has to be performed as well at every node. This is because the encryption operation is orders of magnitude slower than the gradient update.

8.3. Simulating Tree Building. All of our experiments were run on top of the churn trace described in Section 8.1. The network size was 100,000. The membership overlay network was implemented by independently assigning 100 randomly selected outgoing neighbors to each node and then dropping the directionality of the links. This network forms the basis of tree building; the tree neighbors are selected from these nodes. We assume that each node maintains an active TCP connection with its neighbors as suggested in [31]. If a node fails, its neighbors will detect this only with a one-second delay. The neighbor set is constant in our simulations; that is, when a neighbor fails it remains on the list and it is reconnected when it comes back online. The size of our

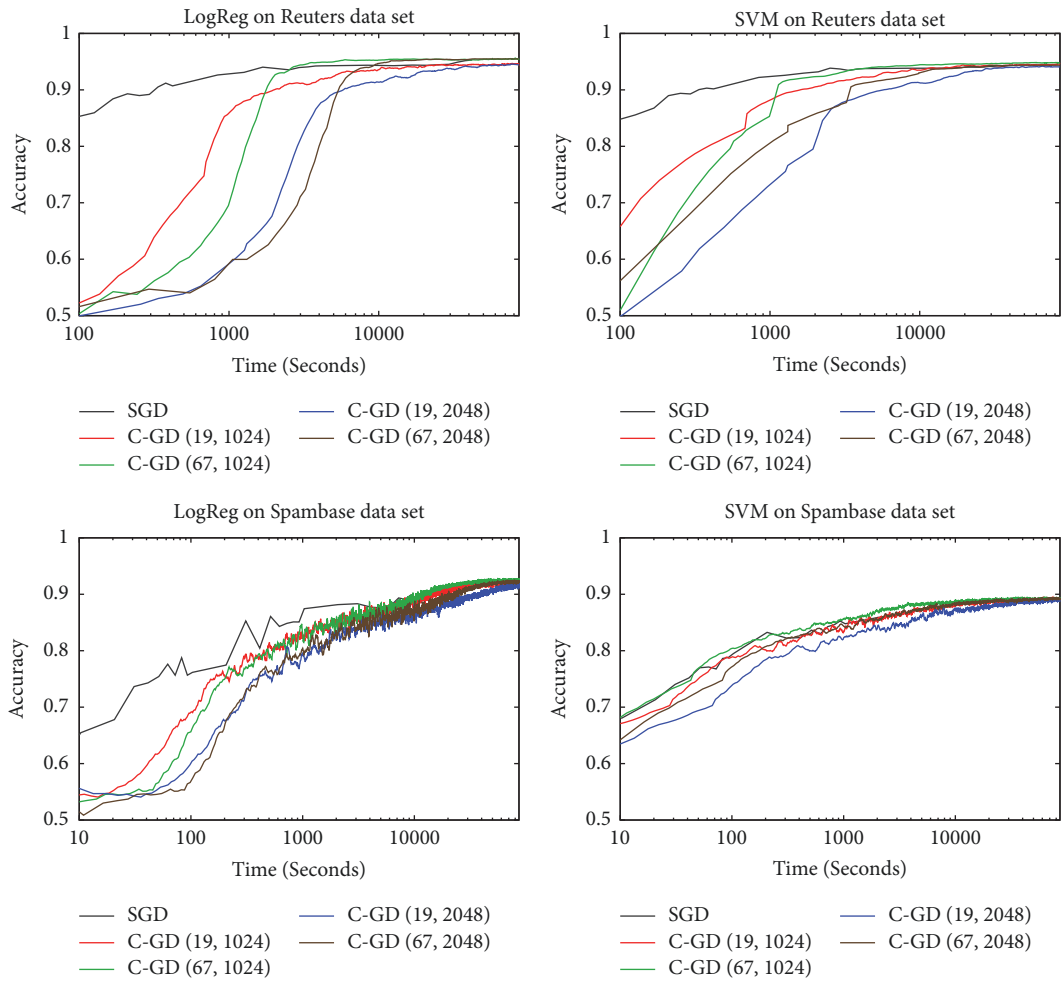

Figure 4: Classification accuracy of the compressed gradient update on the data sets based on trace-based simulation. We vary key size (1024 or 2048) and maximum tree size (19 or 67).

neighbor set was large enough for the overlay network to remain connected.

Initially a random online node is picked from the network at time 0:00 and we simulate building the first tree using that node as root. This simulation involves building the tree and propagating the aggregated gradient up to the root, simulated based on the time consumption of these operations described previously. When this is completed, we pick a new random node that is online at the time of finishing the first minibatch and simulate a new minibatch round. We repeat this procedure until the end of the simulated day. With this methodology, we record the effective minibatch sizes (which determines the number of gradients the sum of which the root actually received) and we examine the distribution of these effective minibatch sizes.

The empirical distributions of the effective minibatch sizes for the case of 10,000 features are shown in Figure 3. In every scenario we simulated a sample of at least 15,000 tree building attempts. The figure shows the histograms based on these samples. The histograms use a logarithmic scale to better illustrate the structure of the distribution. However, note that most of the probability mass belongs to the largest effective sizes. For 100 features almost all the trees are complete due to the very quick building times (not shown). The relatively high probability masses for tree sizes 1, 2, and 3 are due to the vulnerability of the trunk.

In our experiments, we used the variant of the protocol that limits the effective tree size from below as explained in Section 5.5. We accepted a minibatch for gradient update only if its size was greater than or equal to $\lfloor N/2 \rfloor$. The reason is that smaller trees represent reduced privacy. We call such trees a "good tree." The last column of Table 1 contains the probability of getting a good tree. Clearly, only a very small proportion of tree building attempts are unsuccessful.

8.4. Machine Learning Results. We now present our results with the actual learning tasks. The setup for the learning problems is identical to that presented in Section 7. The only difference is that now the batch sizes used in each update step are variable and depend on the effective batch size that is obtained in our tree building simulation based on the smartphone trace, and the time needed to complete a given minibatch is also given by the output of the simulation. The

results are shown in Figure 4. Note that the horizontal axis of the plots now shows the time, covering one full day. It is clear that the main factor for convergence speed is the encryption key size, with 2048 being significantly slower than 1024. This could be expected based on Table 1 as well. We can see that our example learning tasks can converge within one day, which is adequate for many practically interesting learning problems.

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 minibatch method and the sum within the minibatch is calculated using our novel secure algorithm. We can achieve high levels of robustness and good scalability in our tree building protocol through exploiting the fact that the minibatch 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. The algorithm is completely local; therefore it has the same time complexity independently of the network size.

We evaluated the protocol in realistic simulations where we took into account the time needed for encryption and message transmission, and we used a real smartphone trace to simulate churn. We demonstrated on a number of learning tasks that the approach is indeed practically viable even with a key size of 2048. We also demonstrated that the gradients can be compressed by an order of magnitude without sacrificing prediction accuracy.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the Hungarian Government and the European Regional Development Fund under Grant no. GINOP-2.3.2-15-2016-00037 ("Internet of Living Things").

References

[1] J. Konecny, H. B. McMahan, F. X. Yu, P. Richtarik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," in *Private Multi-Party Machine Learning (NIPS 2016 Workshop)*, pp. 1–6, 2016.

[2] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, A. Singh and J. Zhu, Eds., vol. 54, pp. 1273–1282, Machine Learning Research, PMLR, Fort Lauderdale, FL, USA, 2017.

[3] R. Ormandi, I. Hegedus, and M. Jelasity, "Gossip learning with linear models on fully distributed data," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 556–571, 2013.

[4] G. P. Jesi, A. Montresor, and M. Van Steen, "Secure peer sampling," *Computer Networks*, vol. 54, no. 12, pp. 2086–2098, 2010.

[5] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research (JMLR)*, vol. 13, pp. 165–202, 2012.

[6] A. C. Yao, "Protocols for secure computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pp. 160–164, 1982.

[7] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M. Y. Zhu, "Tools for privacy preserving distributed data mining," *ACM SIGKDD Explorations Newsletter*, vol. 4, no. 2, pp. 28–34, 2002.

[8] A. Berta, V. Bilicki, and M. Jelasity, "Defining and understanding smartphone churn over the internet: A measurement study," in *Proceedings of the 14th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2014*, UK, September 2014.

[9] J. Saia and M. Zamani, "Recent results in scalable multi-party computation," in *SOFSEM 2015: theory and practice of computer science*, vol. 8939 of *Lecture Notes in Comput. Sci.*, pp. 24–44, Springer, Heidelberg, 2015.

[10] D. Bickson, T. Reinman, D. Dolev, and B. Pinkas, "Peer-to-peer secure multi-party numerical computation facing malicious adversaries," *Peer-to-Peer Networking and Applications*, vol. 3, no. 2, pp. 129–144, 2010.

[11] J. A. Naranjo, L. G. Casado, and M. Jelasity, "Asynchronous privacy-preserving iterative computation on peer-to-peer networks," *Computing: Archives for Scientific Computing*, vol. 94, no. 8-10, pp. 763–782, 2012.

[12] S. Han, W. K. Ng, L. Wan, and V. C. S. Lee, "Privacy-preserving gradient-descent methods," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 6, pp. 884–899, 2010.

[13] K. Bonawitz, V. Ivanov, B. Kreuter et al., "Practical Secure Aggregation for Privacy-Preserving Machine Learning," in *Proceedings of the the 2017 ACM SIGSAC Conference*, pp. 1175–1191, Dallas, Texas, USA, October 2017.

[14] W. Ahmad and A. Khokhar, "Secure aggregation in large scale overlay networks," in *Proceedings of the IEEE GLOBECOM 2006 - 2006 Global Telecommunications Conference*, USA, December 2006.

[15] C. Dwork, "A firm foundation for private data analysis," *Communications of the ACM*, vol. 54, no. 1, pp. 86–95, 2011.

[16] A. Rajkumar and S. Agarwal, "A differentially private stochastic gradient descent algorithm for multiparty classification," in *Proceedings of the JMLR Workshop and Conference, AISTATS'12*, vol. 22, pp. 933–941, 2012.

[17] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, "Our data, ourselves: privacy via distributed noise generation," in *Advances in cryptology-EUROCRYPT*, vol. 4004 of *Lecture Notes in Comput. Sci.*, pp. 486–503, Springer, Berlin, 2006.

[18] G. Danner and M. Jelasity, "Fully distributed privacy preserving mini-batch gradient descent learning," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 9038, pp. 30–44, 2015.

[19] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger, "On unbiased sampling for unstructured peer-to-peer networks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 2, pp. 377–390, 2009.

[20] U. Maurer, "Secure multi-party computation made simple," *Discrete Applied Mathematics: The Journal of Combinatorial*

Algorithms, Informatics and Computational Sciences, vol. 154, no. 2, pp. 370–381, 2006.

[21] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Identifying suspicious URLs: An application of large-scale online learning," in *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*, pp. 681–688, can, June 2009.

[22] L. Bottou, "Stochastic Gradient Descent Tricks," in *Neural Networks: Tricks of the Trade*, vol. 7700 of *Lecture Notes in Computer Science*, pp. 421–436, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[23] L. Bottou and Y. LeCun, "Large scale online learning," in *Advances in Neural Information Processing Systems 16*, S. Thrun, L. Saul, and B. Scholkopf, Eds., MIT Press, Cambridge, MA, 2004.

[24] K. Gimpel, D. Das, and N. A. Smith, "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.

[25] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in cryptology-EUROCRYPT '99 (Prague)*, vol. 1592 of *Lecture Notes in Comput. Sci.*, pp. 223–238, Springer, Berlin, 1999.

[26] M. Lichman, *UCI machine learning repository*, 2013, http://archive.ics.uci.edu/ml.

[27] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, New York, NY, USA, 2006.

[28] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: primal estimated sub-gradient solver for SVM," *Mathematical Programming*, vol. 127, no. 1, Ser. B, pp. 3–30, 2011.

[29] A. Montresor and M. Jelasity, "PeerSim: A Scalable P2P Simulator," in *Proceedings of the IEEE P2P'09 - 9th International Conference on Peer-to-Peer Computing*, pp. 99-100, USA, September 2009.

[30] *Speedtest: Market reports*, 2017, http://www.speedtest.net/reports/.

[31] R. Roverso, J. Dowling, and M. Jelasity, "Through the wormhole: Low cost, fresh peer sampling for the Internet," in *Proceedings of the 13th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2013*, Italy, September 2013.