BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning

Chengliang Zhang, Suyi Li, Junzhe Xia, and Wei Wang, *Hong Kong University of Science and Technology;* Feng Yan, *University of Nevada, Reno;* Yang Liu, *WeBank*

https://www.usenix.org/conference/atc20/presentation/zhang-chengliang

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning

Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan[†], Yang Liu[‡]
HKUST, [†]University of Nevada, Reno, [‡]WeBank
{czhangbn, slida, jxiaaf, weiwa}@cse.ust.hk, fyan@unr.edu, yangliu@webank.com

Abstract

Cross-silo federated learning (FL) enables organizations (e.g., financial or medical) to collaboratively train a machine learning model by aggregating local gradient updates from each client without sharing privacy-sensitive data. To ensure no update is revealed during aggregation, industrial FL frameworks allow clients to mask local gradient updates using *additively homomorphic encryption* (HE). However, this results in significant cost in *computation* and *communication*. In our characterization, HE operations dominate the training time, while inflating the data transfer amount by two orders of magnitude. In this paper, we present BatchCrypt, a system solution for cross-silo FL that substantially reduces the encryption and communication overhead caused by HE. Instead of encrypting individual gradients with full precision, we *encode a batch of quantized gradients* into a long integer and encrypt it in one go. To allow *gradient-wise aggregation* to be performed on *ciphertexts* of the encoded batches, we develop new quantization and encoding schemes along with a novel gradient clipping technique. We implemented BatchCrypt as a plug-in module in FATE, an industrial cross-silo FL framework. Evaluations with EC2 clients in geo-distributed datacenters show that BatchCrypt achieves 23×-93× training speedup while reducing the communication overhead by 66×-101×. The accuracy loss due to quantization errors is less than 1%.

1 Introduction

Building high-quality machine learning (ML) models requires collecting a massive amount of training data from diverse sources. However, in many industries, data is dispersed and locked in multiple organizations (e.g., banks, hospitals, and institutes), where data sharing is strictly forbidden due to the growing concerns about data privacy and confidentiality as well as violating the government regulations [12, 17, 45]. Cross-silo federated learning (FL) [27, 61] offers an appealing solution to break "data silos" among organizations, where participating clients *collaboratively learn* a global model by uploading their *local gradient updates* to a central server for aggregation, without sharing privacy-sensitive data.

To ensure that no client reveals its update during aggregation, many approaches have been proposed [9, 37, 47, 48, 52]. Among them *additively homomorphic encryption* (HE), notably the Paillier crytosystem [46], is particularly attractive in the cross-silo setting [37,48,61], as it provides a strong privacy guarantee at no expense of learning accuracy loss (§2). With HE, gradient aggregation can be performed on *ciphertexts* without decrypting them in advance. HE has been adopted in many cross-silo FL applications [13, 23, 37, 38, 44], and can be easily plugged into the existing FL frameworks to augment the popular parameter server architecture [33]. Before the training begins, an HE key-pair is synchronized across all clients through a secure channel. During training, each client encrypts its gradient updates using the public key and uploads the ciphertexts to a central server. The server aggregates the encrypted gradients from all clients and dispatches the result to each of them. A client decrypts the aggregated gradients using the private key, updates its local model, and proceeds to the next iteration. As clients only upload the encrypted updates, no information can be learned by the server or an external party during data transfer and aggregation.

Although HE provides a strong privacy guarantee for cross-silo FL, it performs complex cryptographic operations (e.g., modular multiplications and exponentiations) that are extremely expensive to compute. Our testbed characterization (§3) shows that more than 80% of the training iteration time is spent on encryption/decryption. To make matters worse, encryption yields substantially larger ciphertexts, inflating the amount of data transfer by over 150× than plaintext learning. The significant overhead of HE in *encryption* and *communication* has become a major roadblock to facilitating cross-silo FL. According to our contacts at WeBank [57], most of their FL applications cannot afford to use the encrypted gradients and are limited to scenarios with less stringent privacy requirements (e.g., FL across departments or trustworthy partners).

In this paper, we tackle the encryption and communication bottlenecks created by HE with a simple *batch encryption* technique. That is, a client first *quantizes* its gradient values into low-bit integer representations. It then *encodes* a batch of quantized values to a long integer and encrypts it in one go.

Compared with encrypting individual gradient values of full precision, batch encryption significantly reduces the encryption overhead and data transfer amount. Although this idea has been briefly mentioned in the previous work [37, 48], the treatment is rather informal without a viable implementation. In fact, to enable batch encryption in cross-silo FL, there are two key technical challenges that must be addressed, which, to our knowledge, remains open.

First, a feasible batch encryption scheme should allow us to directly sum up the ciphertexts of two batches, and the result, when decrypted, matches that of performing *gradient-wise aggregation* on the two batches in the clear. We show that although it is viable to tweak the generic quantization scheme to meet such need, it has many limitations as it is not designed for aggregation. Instead, we design a customized quantization scheme that quantizes gradient values to *signed integers* uniformly distributed in a *symmetric* range. Moreover, to support gradient-wise aggregation in a simple additive form, and that the addition does not cause overflow to corrupt the encoded gradients, we develop a new batch encoding scheme that adopts *two's compliment representation* with *two sign bits* for quantized values. We also use *padding* and *advance scaling* to avoid overflow in addition. All these techniques allow gradient aggregation to be performed on ciphertexts of the encoded batches, without decryption first.

Second, as gradients values are *unbounded*, they need to be *clipped* before quantization, which critically determines the learning performance [5, 41]. However, it remains unclear how to choose the clipping thresholds in the cross-silo setting. We propose an efficient analytical model dACIQ by extending ACIQ [5], a state-of-the-art clipping technique for ML over centralized data, to cross-silo FL over decentralized data. dACIQ allows us to choose optimal clipping thresholds with the minimum cumulative error.

We have implemented our solution BatchCrypt in FATE [18], a secure computing framework released by WeBank [57] to facilitate FL among organizations. Our implementation can be easily extended to support other optimization schemes for distributed ML such as local-update SGD [22, 35, 56], model averaging [40], and relaxed synchronization [24, 34, 62], all of which can benefit from BatchCrypt when applied to cross-silo FL. We evaluate BatchCrypt with nine participating clients geo-distributed in five AWS EC2 datacenters across three continents. These clients collaboratively learn three ML models of various sizes: a 3-layer fully-connected neural network with FMNIST dataset [60], AlexNet [32] with CIFAR10 dataset [31], and a text-generative LSTM model [25] with Shakespeare dataset [55]. Compared with the stock implementation of FATE, BatchCrypt accelerates the training of the three models by 23×, 71×, and 93×, respectively, where more salient speedup can be achieved for more complex models. In the meantime, the communication overhead is reduced by 66×, 71×, and 101×, respectively. The significant benefits of BatchCrypt come at no cost of model quality, with a negligible accuracy loss less than 1%. BatchCrypt[1] offers the first efficient implementation that enables HE in a cross-silo FL framework with low encryption and communication cost.

2 Background and Related Work

In this section, we highlight the stringent privacy requirements posed by cross-silo federated learning. We survey existing techniques for meeting these requirements.

2.1 Cross-Silo Federated Learning

According to a recent survey [27], federated learning (FL) is a scenario where multiple clients collaboratively train a machine learning (ML) model with the help of a central server; each client transfers local updates to the server for immediate aggregation, without having its raw data leaving the local storage. Depending on the application scenarios, federated learning can be broadly categorized into *cross-device* FL and *cross-silo* FL. In the cross-device setting, the clients are a large number of mobile or IoT devices with limited computing power and unreliable communications [27, 30, 39]. In contrast, the clients in the cross-silo setting are a small number of organizations (e.g., financial and medical) with reliable communications and abundant computing resources in datacenters [27, 61]. We focus on cross-silo FL in this paper.

Compared with the cross-device setting, cross-silo FL has significantly more stringent requirements on privacy and learning performance [27, 61]. *First*, the final trained model should be *exclusively released* to those participating organizations—no external party, including the central server, can have access to the trained model. *Second*, the strong privacy guarantee should not be achieved at a cost of learning accuracy. *Third*, as an emerging paradigm, cross-silo FL is undergoing fast innovations in both algorithms and systems. A desirable privacy solution should impose *minimum constraints* on the underlying system architecture, training mode (e.g., synchronous and asynchronous), and learning algorithms.

2.2 Privacy Solutions in Federated Learning

Many strategies have been proposed to protect the privacy of clients for federated learning. We briefly examine these solutions and comment on their suitability to cross-silo FL.

Secure Multi-Party Computation (MPC) allows multiple parties to collaboratively compute an agreed-upon function with private data in a way that each party knows nothing except its input and output (i.e., zero-knowledge guarantee). MPC utilizes carefully designed computation and synchronization protocols between clients. Such protocols have strong privacy guarantees, but are difficult to implement efficiently

[1] BatchCrypt is open-sourced and can be found at https://github.com/marcoszh/BatchCrypt

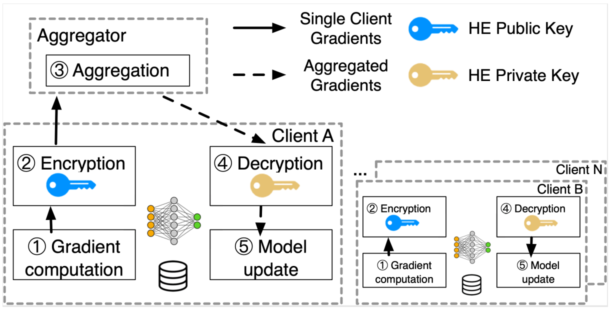


Figure 1: The architecture of cross-silo FL system, where HE is implemented as a pluggable module on the clients.

in a geo-distributed scenario like cross-silo FL [61]. Developers have to carefully engineer the ML algorithms and divide the computation among parties to fit the MPC paradigm, which may lower the privacy guarantees for better performance [16, 42, 43].

Differential Privacy (DP) is another common tool that can be combined with model averaging and SGD to facilitate secure FL [47, 52]. It ensures the privacy of each individual sample in the dataset by injecting noises. A recent work proposes to employ *selective parameter update* [52] atop differential privacy to navigate the tradeoff between data privacy and learning accuracy. Although DP can be efficiently implemented, it exposes plain gradients to the central server during aggregation. Later study shows that one can easily recover the information from gradients [48]. While such privacy breach and the potential accuracy drop might be tolerable for mobile users in cross-device FL, they raise significant concerns for participating organizations in cross-silo FL.

Secure Aggregation [9] is proposed recently to ensure that the server learns no individual updates from any clients but the *aggregated updates* only. While secure aggregation has been successfully deployed in cross-device FL, it falls short in cross-silo FL for two reasons. First, it allows the central server to see the aggregated gradients, based on which the information about the trained model can be learned by an external entity (e.g., public cloud running the central server). Second, in each iteration, clients must synchronize secret keys and *zero-sum masks*, imposing a strong requirement of *synchronous training*.

Homomorphic Encryption (HE) allows certain computation (e.g., addition) to be performed directly on ciphertexts, without decrypting them first. Many recent works [13, 37, 38, 48] advocate the use of additively HE schemes, notably Paillier [46], as the primary means of privacy guarantee in cross-silo FL: each client transfers the encrypted local updates to the server for direct aggregation; the result is then sent back to each client for local decryption. HE meets the three requirements of cross-silo FL. *First*, it protects the trained model from being learned by any external parties including the server as update aggregation is performed on ciphertexts. *Second*, it incurs no learning accuracy loss, as no noise is added to the model updates during the encryption/decryption process. *Third*, HE directly applies to the existing learning systems, requiring no modifications other than encrypting/decrypting updates. It hence imposes no constraints to the synchronization schemes and the learning algorithms. However, as we shall show in §3, HE introduces significant overhead to computation and communication.

Summary To summarize, each of these privacy-preserving techniques has its pros and cons. MPC is able to provide strong privacy guarantees, but requires expert efforts to re-engineer existing ML algorithms. DP can be adopted easily and efficiently, but has the downside of weaker privacy guarantee and potential accuracy loss. Secure aggregation is an effective way to facilitate large-scale cross-device FL, but may not be suitable for cross-silo FL as it exposes the aggregated results to third parties and incurs high synchronization cost. HE can be easily adopted to provide strong privacy guarantees without algorithm modifications or accuracy loss. However, the high computation and communication overheads make it impractical for production deployment at the moment.

2.3 Cross-Silo FL Platform with HE

Fig. 1 depicts a typical cross-silo FL system [27, 37, 61], where HE is implemented as a pluggable module on the clients. The *aggregator* is the server which coordinates the *clients* and aggregates their encrypted gradients. Note that in this work, we assume the aggregator is *honest-but-curious*, a common threat model used in the existing FL literature [9, 38, 52]. The communications between all parties (the clients and the aggregator) are secured by cryptographic protocols such as SSL/TLS, so that no third party can learn the messages being transferred. Before the training starts, the aggregator randomly selects a client as the leader who generates an HE key-pair and synchronizes it to all the other clients. The leader also initializes the ML model and sends the model weights to all the other clients. Upon receiving the HE key-pair and the initial weights, the clients start training. In an iteration, each client computes the local gradient updates (①), encrypts them with the public key (②), and transfers the results to the aggregator. The aggregator waits until the updates from all the clients are received. It then adds them up and dispatches the results to all clients (③). A client then decrypts the aggregated gradients (④) and uses it to update the local model (⑤).

This architecture design follows the classic distributed SGD pattern. So the existing theories and optimizations including flexible synchronization [24, 34, 62] and local update SGD [22, 35, 56] naturally apply. Moreover, as model updating is performed on the client's side using the plaintext gradient aggregation, we can adopt state-of-the-art adaptive optimizers such as Adam [28] for faster convergence—a huge advantage over the existing proposal [48] that applies encrypted gradients directly on the encrypted global model in the server.

3 Characterizing Performance Bottlenecks

In this section, we characterize the performance of cross-silo FL with three real applications driven by deep learning models in a geo-distributed setting. We show that encryption and communication come as two prohibitive bottlenecks that impede the adoption of FL among organizations. We survey possible solutions in the literature and discuss their inefficiency. To our knowledge, we are the first to present a comprehensive characterization for cross-silo FL in a realistic setting.

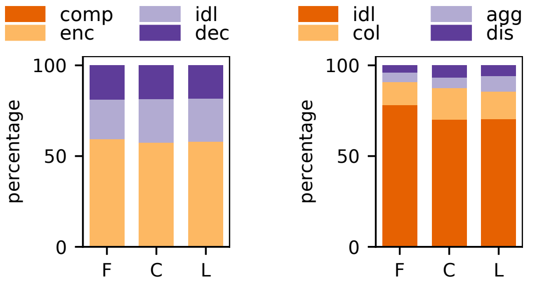
3.1 Characterization Results

Cross-silo FL is usually performed in multiple *geo-distributed* datacenters of participating organizations [27, 61]. Our characterization is carried out in a similar scenario where nine EC2 clients in five geo-distributed datacenters collaboratively training three ML models of various sizes, including FMNIST, CIFAR, and LSTM (Table 3). Unless otherwise specified, we configure *synchronous training*, where no client can proceed to the next iteration until the (encrypted) updates from all clients have been aggregated. We defer the detailed description of the cluster setup and the ML models to §6.1.

We base our study in FATE (Federated AI Technology Enabler) [18], a secure compute framework developed by WeBank [57] to drive its FL applications with the other industry partners. To our knowledge, FATE is the only open-source cross-silo FL framework deployed in production environments. FATE has a built-in support to the Pailler cryptosystem [46] (key size set to 2048 bits by default), arguably the most popular additively HE scheme [50]. Our results also apply to the other partially HE cryptosystems.

Encryption and Communication Overhead We start our characterization by comparing two FL scenarios, with and without HE. We find that the use of HE results in *exceedingly long training time* with *dramatically increased data transfer*. More specifically, when HE is enabled, we measured the average training iteration time 211.9s, 2725.7s, and 8777.7s for FMNIST, CIFAR, and LSTM, respectively. Compared with directly transferring the plaintext updates, the iteration time is extended by 96×, 135×, and 154×, respectively. In the meantime, when HE is (not) in use, we measured 1.1GB (6.98MB), 13.1GB (85.89MB), and 44.1GB (275.93MB) data transfer between clients and aggregator in one iteration on average for FMNIST, CIFAR, and LSTM, respectively. To sum up, the use of HE increases both the training time and the network footprint by two orders of magnitude. Such performance overhead becomes even more significant for complex models with a large number of weights (e.g., LSTM).

Deep Dive To understand the sources of the significant overhead caused by HE, we examine the training process of the three models in detail, where we sample an iteration and depict in Fig. 2 the breakdown of the iteration time spent on different operations on the client's side (left) and on the aggregator's side (right), respectively.



(a) Client: **comp**ute, **enc**rypt, **idl**e, **dec**rypt

(b) Aggregator: **idl**e, **col**lect, **agg**regate, **dis**patch

Figure 2: Iteration time breakdowns of **F**MNIST, **C**IFAR, and **L**STM for a client and the aggregator.

As illustrated in Fig. 2a, on the client's side, HE-related operations dominate the training time in all three applications. In particular, a client spent around 60% of the iteration time on gradient encryption (yellow), 20% on decryption (dark purple), and another 20% on data transfer and idle waiting for the gradient aggregation to be returned[2] (light purple). In comparison, the time spent on the actual work for computing the gradients becomes *negligible* ($< 0.5\%$).

When it comes to the aggregator (Fig. 2b), most of the time ($> 70\%$) is wasted on idle waiting for a client to send in the encrypted gradients (orange). Collecting the gradients from all clients (yellow) and dispatching the aggregated results to each party (dark purple) also take a significant amount of time, as clients are geo-distributed and may not start transferring (or receiving) at the same time. The actual computation time for gradient aggregation (light purple) only accounts for less than 10% of the iteration span. Our deep-dive profiling identifies encryption and decryption as the two dominant sources of the exceedingly long training time.

Why is HE So Expensive? In additively HE cryptosystems such as Paillier [46], encryption and decryption both involve multiple modular multiplications and exponentiation operations with a large exponent and modulus (usually longer than 512 bits) [50], making them extremely expensive to compute. Encryption also yields significantly larger ciphertexts, which, in turn, causes a huge communication overhead for data transfer. In additively HE schemes such as Paillier, a ciphertext takes roughly the same number of bits as the key size, irrespective of the plaintext size. As of 2019, the minimum secure key size for Paillier is 2048 [6], whilst a gradient is typically a 32-bit floating point. This already translates to 64× size inflation after encryption.

We further benchmark the computation overhead and the inflated ciphertexts of Paillier with varying key sizes. We use `python-paillier` [15] to encrypt and then decrypt 900K 32-bit floating points. Table 1 reports the results on

[2] Due to the synchronization barrier, a client needs to wait for all the other clients to finish transferring updates to the aggregator.

Table 1: Benchmarking Paillier HE with various key sizes.

Key size	Plaintext	Ciphertext	Encryption	Decryption
1024	6.87MB	287.64MB	216.87s	68.63s
2048	6.87MB	527.17MB	1152.98s	357.17s
3072	6.87MB	754.62MB	3111.14s	993.80s

a `c5.4xlarge` instance. As the key size increases (higher security), both the computation overhead and the size of ciphertexts grow linearly. Since Paillier can only encrypt integers, floating point values have to be scaled beforehand, and their exponents information contribute further to data inflation.

Summary The prohibitive computation and communication overhead caused by HE, if not properly addressed, would lead to two serious economic consequences. First, given the dominance of HE operations, accelerating model computation using high-end hardware devices (e.g., GPUs and TPUs) is no longer relevant—a huge waste of the massive infrastructure investments in clients' datacenters. Second, the overwhelming network traffics across geo-distributed datacenters incurs skyrocketing Internet data charges, making cross-silo FL economically unviable. In fact, in WeBank, production FL applications may choose to turn off HE if the security requirement is not so strict.

3.2 Potential Solutions and Their Inefficiency

Hardware-Accelerated HE HE process can be accelerated using software or hardware solutions. However, typical HE cryptosystems including Paillier have limited interleaving independent operations, thus the potential speedup of a single HE operation is quite limited. In fact, it is reported that a specialized FPGA can only accelerate Paillier encryption by 3× [50]. Moreover, simply accelerating the encryption itself does not help reduce the communication overhead.

Reducing Communication Overhead As accelerating HE itself does not clear the barrier of adopting HE in FL, what if we reduce the amount of data to encrypt in the first place? Since data inflation is mainly caused by the mismatch between the lengths of plaintexts and ciphertexts, an intuitive idea would be *batching* as many gradients together as possible to form a *long* plaintext, so that the amount of encryption operations will reduce greatly. However, the challenge remains how to maintain HE's additive property after batching without modifying ML algorithms or hurting the learning accuracy.

While some prior works have explored the idea of joining multiple values together to reduce HE overhead, they give no viable implementation of batch encryption for cross-silo FL. [48] makes a false assumption that quantization is lossless, and uses adaptive optimizer Adam in its simulation even though its design does not support that. With only plain SGD available, [48] requires tedious learning rate scheduling tuning to achieve similar results of advanced optimizers [59]. The naive batching given in [37] cannot be correctly implemented as homomorphic additivity is not retained. In fact, none of these works have systematically studied the impact of batching. Gazelle [26] and SEAL [51] adopt the SIMD (single instruction multiple data) technique to speed up HE. However, such approach only applies to lattice-based HE schemes [11] and is restricted by their unique properties. For instance, it incurs dramatic computational complexity for lattice-based HE schemes to support more levels of multiplication [26]. Besides, these works only accelerate integer cryptographic operations. How to maintain the training accuracy in cross-silo FL context remains an open problem.

4 BatchCrypt

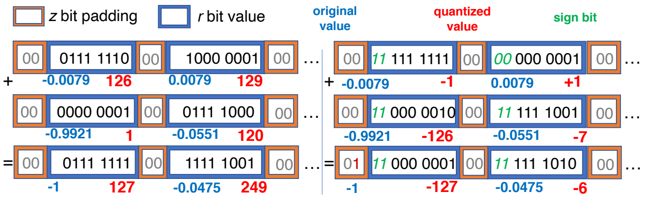
In this section, we describe our solution for gradient batching. We begin with the technical challenges. We first show that gradient quantization is required to enable batching. We then explain that generic quantization scheme lacks flexibility and efficiency to support general ML algorithms, which calls for an appropriately designed encoding and batching scheme; to prevent model quality degradation, an efficient clipping method is also needed. We name our solution BatchCrypt, a method that co-designs quantization, batch encoding, and analytical quantization modeling to boost computation speed and communication efficiency while preserving model quality in cross-silo FL with HE.

4.1 Why is HE Batching for FL a Problem?

On the surface, it seems straightforward to implement gradient batching. In fact, batching has been used to speed up queries over integers in a Paillier-secured database [19]. However, this technique only applies to *non-negative integers* [19]. In order to support floating numbers, the values have to be *reordered and grouped by their exponents* [19]. Such constraints are the key to preserving HE's additivity of batched ciphertexts—that is, the sum of two batched ciphertexts, once decrypted, should match the results of element-wise adding plaintext values in the two groups. Gazelle and SEAL [26, 51] employ SIMD technique to meet this requirement, but the approach is limited to lattice-based cryptosystems. We aspire to propose a universal batching method for all additively homomorphic cryptosystems.

Why Quantization is Needed? Gradients are *signed floating values* and must be ordered by their corresponding model weights, for which we cannot simply rearrange them by exponents. The only practical approach is to use integer representations of gradients in the batch, which requires quantization.

Existing Quantization Schemes ML algorithms are resilient to update noise and able to converge with gradients of limited precision [10]. Fig. 3a illustrates how generic gradient quantization scheme can be used in HE batching. Notably, since there is no bit-wise mapping between a ciphertext and its plaintext, permutation within ciphertexts is not allowed—only plain bit-by-bit addition between batched integers is available. Assume a gradient g in $[-1,1]$ is quantized into an



(a) generic quantization

(b) BatchCrypt

Figure 3: An illustration of a generic quantization scheme and BatchCrypt. The latter preserves additivity during batching, with the sign bits highlighted within values.

8-bit unsigned integer. Let $[\cdot]$ denote the standard rounding function. The quantized value of g is

$$Q(g) = [255 * (g - min)/(max - min)],$$

where $max = 1$ and $min = -1$. Suppose n quantized gradients are summed up. The result, denoted by q_n, is dequantized as

$$Q^{-1}(q_n) = q_n * (max - min)/255 + n * min.$$

Referring to Fig. 3a, gradients of a client (floating numbers in blue) are first quantized and then batch joined into a large integer. To aggregate the gradients of two clients, we simply sum up the two batched integers, locate the added gradients at the same bit positions as in the two batches (8-bit integers in red), and dequantize them to obtain the aggregated results.

Such a generic quantization scheme, though simple to implement, does not support aggregation well and has many ***limitations*** when applied to batched gradient aggregation.

(1) It is restrictive. In order to dequantize the results, it must know how many values are aggregated. This poses extra barriers to flexible synchronization, where the number of updates is constantly changing, sometimes even unavailable.

(2) It overflows easily in aggregation. As values are quantized into positive integers, aggregating them is bound to overflow quickly as the sum grows larger. To prevent overflow, batched ciphertexts have to be decrypted after a few additions and encrypted again in prior work [48].

(3) It does not differentiate *positive* overflows from *negative*. Once overflow occurs, the computation has to restart. Should we be able to tell them apart, a saturated value could have been used instead of discarding the results.

4.2 HE Batching for Gradients

Unsatisfied with the generic quantization technique, we aspire to devise a batching solution tailored to gradient aggregation. Our scheme should have the following desirable properties: (1) it preserves the additivity of HE; (2) it is more resilient to overflows and can distinguish positive overflows from negative ones; (3) it is generally applicable to existing ML algorithms and optimization techniques; (4) it is flexible enough that one can dequantize values directly without additional information, such as the number of values aggregated.

Gradient Quantization Existing works use gradient compression techniques to reduce network traffic in distributed training [1,29,36,58]. These quantization methods are mainly used to compress values for transmission [58] or accelerate inference where only multiplication is needed [5]. However, they are not designed for gradient aggregation, and we cannot perform computations over the compressed gradients efficiently, making them inadequate for FL. We scrutinize the constraints posed by our design objectives, and summarize the stemed requirements for quantization as follows:

- **Signed Integers:** Gradients should be quantized into *signed* integers. In this way, positive and negative values can cancel each other out in gradient aggregation, making it less prone to overflowing than quantizing gradients into unsigned integers.
- **Symmetric Range:** To make values with opposite signs cancel each other out, the quantized range must be symmetrical. Violating this requirement may lead to an incorrect aggregation result. For example, if we map $[-1,1]$ to $[-128,127]$, then $-1+1$ would become $-128+127 = -1$ after quantization.
- **Uniform Quantization:** Literature shows that non-uniform quantization schemes have better compression rates as gradients have non-uniform distribution [1,7]. However, we are unable to exploit the property as additions over quantized values are required.

BatchCrypt We now propose an efficient quantization scheme BatchCrypt that meets all the requirements above. Assume that we quantize a gradient in $[-\alpha,\alpha]$ into an r-bit integer. Instead of mapping the whole range all together, we *uniformly map* $[-\alpha,0]$ and $[0,\alpha]$ to $[-(2^r-1),0]$ and $[0,2^r-1]$, respectively. Note that the value 0 ends up with two codes in our design. Prior work shows that 16-bit quantization ($r = 16$) is sufficient to achieve near lossless gradient quantization [21]. We will show in §6 that such a moderate quantization width is sufficient to enable efficient batching in FL setting.

With quantization figured out, the challenge remains how to encode the quantized values so that signed additively arithmetic is correctly enabled—once the batched long integer is encrypted, we cannot distinguish the sign bits from the value bits during aggregation. Inspired by how modern CPUs handle signed integer computations, we use *two's complement representation* in our encoding. By doing so, the sign bits can engage in the addition just like the value bits. We further use the *two sign bits* to differentiate between the positive and negative overflows. We illustrate an example of BatchCrypt in Fig. 3b. By adding the two batched long integers, BatchCrypt gets the correct aggregation results for $-1+(-126)$ and $+1+(-7)$, respectively.

BatchCrypt achieves our requirements by co-designing quantization and encoding: no additional information is needed to dequantize the aggregated results besides the batch itself; positive and negative values are able to offset each other; the signs of overflow can be identified. Compared

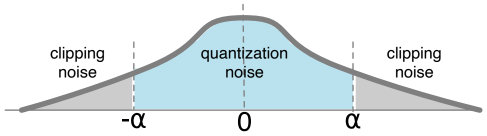


Figure 4: A typical layer gradient distribution. α is the clipping threshold.

with the batching methods in [26, 51], BatchCrypt's batching scheme is generally applicable to all additively HE cryptosystems' and fully HE cryptosystems' additive operations.

4.3 dACIQ: Analytical Clipping for FL

Our previous discussion has assumed gradients in a bounded range (§4.2). In practice, however, gradients may go unbounded and need to be *clipped* before quantization. Also, gradients from different layers have different distributions [58]. We thus need to quantize layers individually [1, 58]. Moreover, prior works show that gradients from the same layer have a bell-shaped distribution which is near Gaussian [2, 7, 53]. Such property can be exploited for efficient gradient compression [1, 58]. Finally, gradients require *stochastic rounding* during quantization [21, 36, 58], as it stochastically preserves diminishing information compared to *round-to-nearest*.

Layer-wise quantization and stochastic rounding can be easily applied, yet it remains unclear how to find the optimal clipping thresholds in the FL setting. As shown in Fig. 4, clipping is the process of saturating the outlaying gradients beyond a threshold α. If α is set too large, the quantization resolution becomes too low. On the other hand, if α gets too small, most of the range information from outlaying gradients has to be discarded.

In general, there are two ways to set the clipping threshold, *profiling-based* methods and *analytical modeling*. Profiling-based clipping selects a sample dataset to obtain a sample gradient distribution. Thresholds are then assessed with metrics such as KL divergence [41] and convergence rate [58]. However, such approach is impractical in FL for three reasons. First, finding a representative dataset in FL can be difficult, as clients usually have non-i.i.d. data, plus it breaks the data silo. Second, the gradient range narrows slowly as the training progresses [14], so clipping needs to be calibrated constantly, raising serious overhead concerns. Third, the profiling results are specific to the training models and datasets. Once the models or the datasets change, new profiling is needed. For both practicality and cost considerations, BatchCrypt instead adopts analytical modeling.

As shown in Fig. 4, the accumulated noise comes from two sources. *Quantization noise* refers to the error induced by rounding within the clipping range (the light blue area), while *clipping noise* refers to the saturated range beyond the clipping threshold (the gray area). To model the accumulated noise from both quantization and clipping, state-of-the-art clipping technique ACIQ [5] assumes that they follow a Gaussian distribution. However, ACIQ cannot be directly applied to BatchCrypt for two reasons. First, it employs a generic asymmetric quantization, which is not the case in BatchCrypt; second, in FL, gradients are not available at one place in plaintext to conduct distribution fitting.

We address these problems by extending ACIQ clipping to the distributed FL setting, which we call dACIQ. In particular, we adopt stochastic rounding with an r-bit quantization width. Assume that gradients follow Gaussian distribution $X \sim N(0, \sigma^2)$. Let q_i be the i-th quantization level. We compute the accumulated error in BatchCrypt as follows:

$$\begin{aligned} E[(X-Q(X))^2] &= \int_{-\infty}^{-\alpha} f(x)\cdot(x+\alpha)^2 dx + \int_{\alpha}^{\infty} f(x)\cdot(x-\alpha)^2 dx \\ &+ \sum_{i=0}^{2^r-3} \int_{q_i}^{q_{i+1}} f(x)\ \cdot [\ (x-q_i)^2 \cdot (\frac{q_{i+1}-x}{\triangle})\ +\ (x-q_{i+1})^2 \cdot (\frac{x-q_i}{\triangle})\ ]dx \\ &\approx \frac{\alpha^2+\sigma^2}{2}\cdot[1-erf(\frac{\alpha}{\sqrt{2}\sigma})] - \frac{\alpha\cdot\sigma\cdot e^{-\frac{\alpha^2}{2\cdot\sigma^2}}}{\sqrt{2\pi}} + \frac{2\alpha^2\cdot(2^r-2)}{3\cdot 2^{3r}}, \end{aligned} \tag{1}$$

where the first and the second terms account for the clipping noise, and the third the rounding noise. As long as we know σ, we can then derive the optimal threshold α from Eq. (1). We omit the detailed derivations in the interest of space.

Gaussian Fitting Now that we have Eq. (1), we still need to figure out how to fit gradients into a Gaussian distribution in the FL setting. Traditionally, to fit Gaussian parameters μ and σ, Maximum Likelihood Estimation and Bayesian Inference can be used. They require information including the size of observation set, its sum, and its sum of squares. As an ML model may have up to millions of parameters, calculating these components as well as transferring them over Internet is prohibitively expensive. As a result, dACIQ adopts a simple, yet effective Gaussian fitting method proposed in [4]. The method only requires the size of observation set and its max and min, with the minimum computational and communication overhead. We later show that such light-weight fitting does not affect model accuracy in §6.

Advance Scaling With multiple clients in FL, it is essential to prevent overflows from happening. Thanks to clipping, the gradient range is predetermined before encryption. Let m be the number of clients. If m is available, we could employ *advance scaling* by setting the quantization range to m times of the clipping range, so that the sum of gradients from all clients will not overflow.

4.4 BatchCrypt: Putting It All Together

Putting it all together, we summarize the workflow of BatchCrypt in Algorithm 1.

Initialization The aggregator randomly selects one client as the leader. The leader client generates the HE key-pair and initializes the model weights. The key-pair and model weights are then synchronized with the other client workers.

Training After initialization, there is no differentiation between the leader and the other workers. Clients compute gra-

Algorithm 1 HE FL BatchCrypt

```
Aggregator:
function INITIALIZE
    Issue INITIALIZELEADER() to the randomly selected leader
    Issue INITIALIZEOTHER() to the other clients
function STARTSTRAINING
    for epoch $e = 0, 1, 2, ..., E$ do
        Issue WORKERSTARTSEPOCH($e$) to all clients
        for all training batch $t = 0, 1, 2, \cdots, T$ do
            Collect gradients range and size
            Return clipping values $\alpha$ calculated by dACIQ
            Collect, sum up all $g_i^{(e,t)}$ into $g^{(e,t)}$, and dispatch it

Client Worker: $i = 1, 2, \ldots, m$
    – $r$: quantization bit width, $bs$: BatchCrypt batch size
function INITIALIZELEADER
    Generate HE key-pair pub_key and pri_key
    Initialize the model to train w
    Send pub_key, pri_key, and w to other clients
function INITIALIZEOTHER
    Receive HE key-pair pub_key and pri_key
    Receive the initial model weights w
function WORKERSTARTSEPOCH($e$)
    for all training batch $t = 0, 1, 2, \cdots, T$ do
        Compute gradients $g_i^{(e,t)}$ based on $w$
        Send per-layer range and size of $g_i^{(e,t)}$ to aggregator
        Receive the layer-wise clipping values $\alpha$'s
        Clip $g_i^{(e,t)}$ with corresponding $\alpha$, quantize $g_i^{(e,t)}$ into $r$ bits, with
    quantization range setting to $m\alpha$                    ▷ Advance scaling
        Batch $g_i^{(e,t)}$ with $bs$ layer by layer
        Encrypt batched $g_i^{(e,t)}$ with pri_key
        Send encrypted $g_i^{(e,t)}$ to aggregator
        Collect $g^{(e,t)}$ from aggregator, and decrypt with pub_key
        Apply decrypted $g^{(e,t)}$ to $w$
```

dients and send the per-layer gradient range and size to the aggregator. The aggregator estimates the Gaussian parameters first and then calculates the layer-wise clipping thresholds as described in § 4.3. Clients then quantize the gradients with range scaled by the number of clients, and encrypt the quantized values using BatchCrypt. Note that advanced scaling utilizing the number of clients is used to completely avoid overflowing. However, Algorithm 1 is still viable even without that information, as BatchCrypt supports overflow detection. The encrypted gradients are gathered at the aggregator and summed up before returning to the clients.

5 Implementation

We have implemented BatchCrypt atop FATE (v1.1) [18]. While we base our implementation on FATE, nothing precludes it from being extended to the other frameworks such as TensorFlow Federated [20] and PySyft [49].

Overview Our implementation follows the paradigm described in Algorithm 1, as most of the efforts are made on the client side. Fig. 5 gives an overview of the client architecture.

BatchCrypt consists of dACIQ, Quantizer, two's Compliments Codec, and Batch Manager. dACIQ is responsible for Gaussian fitting and clipping threshold calculation. Quantizer takes the thresholds and scales them to quantize the clipped values into signed integers. Quantizer also performs dequantization. Two's Compliments Codec translates between a quantized value's true form and two's compliment form with two sign bits. Given the large volume of data to encode, we adopt Numba to enable faster machine codes and massive parallelism. Finally, Batch Manager is in charge of batching and unbatching gradients in their two's compliment form, it remembers data's original shape before batching and restores it during unbatching. Batch Manager utilizes joblib to exploit computing resources by multiprocessing. FATE is used as an infrastructure to conduct FL, in which all the underlying ML computations are written with TensorFlow v1.14 optimized for our machines shipped with AWS DLAMI [3]. FATE adopts the open-sourced python-paillier as the Paillier HE implementation. We again employ joblib to parallel the operations here. FATE's Communication Manager conducts the SSL/TLS secured communication with gRPC. During our characterizations and evaluations, the CPUs are always fully utilized during Paillier operations and BatchCrypt process.

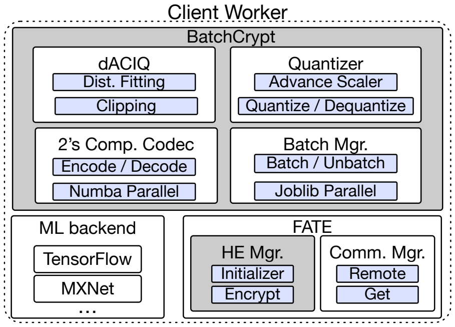


Figure 5: The architecture of a client worker in BatchCrypt.

Model Placement In the typical parameter server architecture, model weights are placed on the server side, while we purposely place weights on the worker side in BatchCrypt. Prior work [48] employs the traditional setup: clients encrypt the initialized weights with HE and send them to the aggregator first; the aggregator applies the received encrypted gradients to the weights encrypted with the same HE key. Such placement has two major drawbacks. First, keeping weights on the aggregator requires *re-encryption*. Since new gradients are constantly applied to weights, the model has to be sent back to the clients to decrypt and re-encrypt to avoid overflows from time to time, resulting in a huge overhead. Second, applying encrypted gradients prevents the use of sophisticated ML optimizers. State-of-the-art ML models are usually trained with adaptive optimizers [28] that scale the learning rates according to the gradient itself. By keeping the model weights on the client side, BatchCrypt can examine the aggregated plaintext gradients, enabling the use of advanced optimizers like Adam, whereas on the aggregator side, one can only adopt plain SGD.

Table 2: Network bandwidth (Mbit/sec) between aggregator and clients in different regions.

Region	Ore.	TYO.	N.VA.	LDN	HK
Uplink (Mbps)	9841	116	165	97	81
Downlink (Mbps)	9841	122	151	84	84

6 Evaluation

In this section, we evaluate the performance of BatchCrypt with real ML models trained in geo-distributed datacenters. We first examine the learning accuracy loss caused by our quantization scheme (§6.2). We then evaluate the computation and communication benefits BatchCrypt brings as well as how its performance compares to the ideal plaintext learning (§6.3). We then assess how BatchCrypt's speedup may change with various batch sizes (§6.4). Finally, we demonstrate the significant cost savings achieved by BatchCrypt (§6.5).

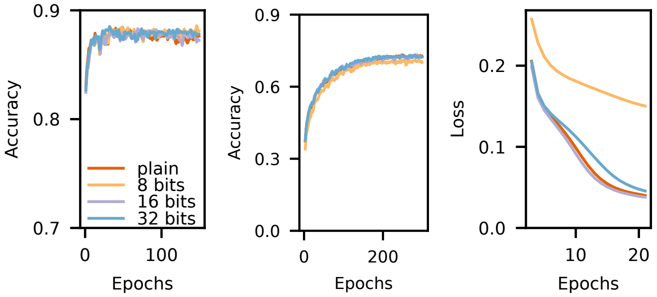
6.1 Methodology

Setting We consider a geo-distributed FL scenario where nine clients collaboratively train an ML model in five AWS EC2 datacenters located in Tokyo, Hong Kong, London, N. Virginia, and Oregon, respectively. We launched two compute-optimized `c5.4xlarge` instances (16 vCPUs and 32 GB memory) as two clients in each datacenter except that in Oregon, where we ran only one client. Note that we opt to not use GPU instances because computation is not a bottleneck. We ran one aggregator in the Oregon datacenter using a memory-optimized `r5.4xlarge` instance (16 vCPUs and 128 GB memory) in view of the large memory footprint incurred during aggregation. To better outline the network heterogeneity caused by geo-locations, we profiled the network bandwidth between the aggregator and the client instances. Our profiling results are summarized in Table 2. We adopt Pailler cryptosystem in our evaluation as it is widely adopted in FL [50], plus batching over it is not supported by Gazelle or SEAL [26, 51]. We expect our results also apply to other cryptosystems as BatchCrypt offers a generic solution.

Benchmarking Models As there is no standard benchmarking suites for cross-silo FL, we implemented three representative ML applications in FATE v1.1. Our first application is a 3-layer fully-connected neural network trained over FMNIST dataset [60], where we set the training batch size to 128 and adopt Adam optimizer. In the second application, we train AlexNet [32] using CIFAR10 dataset [31], with batch size 128 and RMSprop optimizer with 10^{-6} decay. The third application is an LSTM model [25] with Shakespeare dataset [55], where we set the batch size to 64 and adopt Adam optimizer. Other LSTM models that are easier to validate have significantly more weights. Training them to convergence is beyond our cloud budget. As summarized in Table 3, all three applications are backed by deep learning models of various sizes and cover common learning tasks such as image classification and text generation. For each application, we randomly partition its training dataset across nine clients. We configure synchronous training unless otherwise specified.

Table 3: Summary of models used in characterizations.

	FMNIST	**CIFAR**	**LSTM**
Network	3-layer FC	AlexNet [32]	LSTM [25]
Weights	101.77K	1.25M	4.02M
Dataset	FMNIST [60]	CIFAR10 [31]	Shakespeare [55]
Task	Image class.	Image class.	Text generation



(a) FMNIST test acc. (b) CIFAR test acc. (c) LSTM train loss

Figure 6: The quality of trained model with different quantization bit widths in BatchCrypt.

6.2 Impact of BatchCrypt's Quantization

We first evaluate the impact of our quantization scheme, and see how quantization bit width could affect the model quality. We report the test accuracy for FMNIST and CIFAR workloads to see how BatchCrypt's quantization affects the classification top-1 accuracy. Training loss is used for LSTM as the dataset is unlabelled and has no test set. We simulated the training with nine clients using BatchCrypt's quantization scheme including dACIQ clipping. The simulation scripts are also open-sourced for public access. We set the quantization bit width to 8, 16, and 32, respectively, and compare the results against plain training (no encryption) as the baseline. We ran the experiments until convergence, which is achieved when the accuracy or loss does not reach a new record for three consecutive epochs.

Fig. 6 depicts the results. For FMNIST, plain baseline reaches peak accuracy 88.62% at the 40th epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 88.67%, 88.37%, and 88.58% at the 122nd, 68th, and 32nd epoch, respectively. For CIFAR, plain baseline reaches peak accuracy 73.97% at the 285th epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 71.47%, 74.04%, and 73.91% at the 234th, 279th, and 280th epoch, respectively. Finally, for LSTM, plain baseline reaches bottom loss 0.0357 at the 20th epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 0.1359, 0.0335, and 0.0386 at the 29th, 23rd, and 22nd epoch, respectively. We hence conclude that, with appropriate quantization bit width, BatchCrypt's quantization has negligible negative impact on the trained model quality. Even in

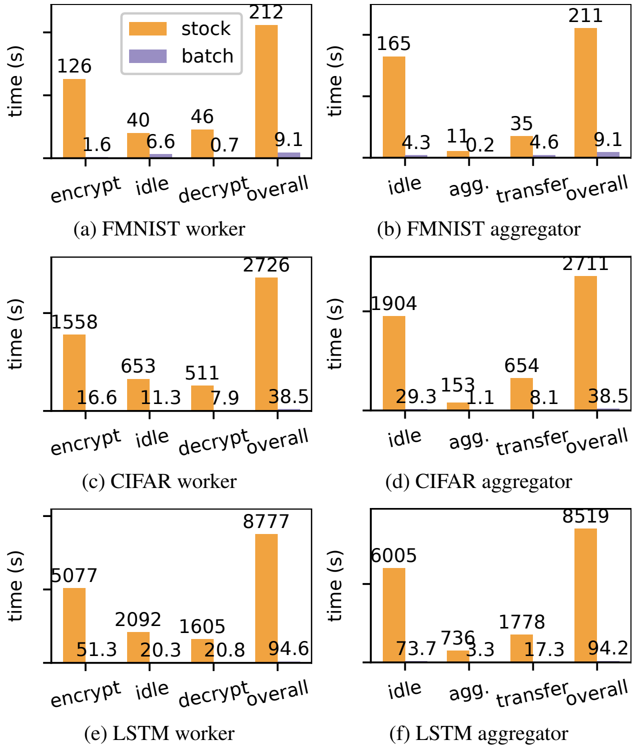


Figure 7: Breakdown of training iteration time under stock FATE and BatchCrypt, where "idle" measures the idle waiting time of a worker and "agg." measures the gradient aggregation time on the aggregator. Note that model computation is left out here as it contributes little to the iteration time.

the case where the quantized version requires more epochs to converge, we later show that such overhead can be more than compensated by the speedup from BatchCrypt.

Although 8-bit quantization performs poorly for CIFAR and LSTM, it is worth notice that, longer bit width does not necessarily lead to higher model quality. In fact, quantized training sometimes achieves better results. Prior quantization work has observed similar phenomenon [63], where the stochasticity introduced by quantization can work as a regularizer to reduce overfitting, similar to a dropout layer [54]. Just like the dropout rate, quantization bit width acts as a trade-off knob for how much information is retained and how much stochasticity is introduced.

In summary, with apt bit width, our gradient quantization scheme does not adversely affect the trained model quality. In contrast, existing batching scheme introduces 5% of quality drop [37]. Thus, quantization-induced error is not a concern for the adoption of BatchCrypt.

6.3 Effectiveness of BatchCrypt

BatchCrypt vs. FATE We next evaluate the effectiveness of BatchCrypt in real deployment. We set the quantization bit width to 16 as it achieves a good performance (§6.2). The batch size is set to 100, in which we pad two zeros between the two adjacent values. We report two metrics: the iteration time breakdown together with the network traffic. We ran the experiments for 50 iterations, and present the averaged results against those measured with the stock FATE implementation in Figs. 7 and 8. We see in Fig. 2 that BatchCrypt significantly speeds up a training iteration: 23.3× for FMNIST, 70.8× for CIFAR, and 92.8× for LSTM. Iteration time breakdown further shows that our implementation reduces the cost of HE related operations by close to 100×, while the communication time is substantially reduced as well ("idle" in worker and "transfer" in aggregator).

We next refer to Fig. 8, where we see that BatchCrypt reduces the network footprint by up to 66×, 71×, and 101× for FMNIST, CIFAR, and LSTM, respectively. Note that FATE adopts `grpc` as the communication vehicle whose limit on payload forces segmenting encrypted weights into small chunks before transmission. By reducing the size of data to transfer, BatchCrypt alleviates the segmenting induced overhead (metadata, checksum, etc.), so it is possible to observe a reduction greater than the batch size.

Our experiments also show that BatchCrypt achieves more salient improvements for larger models. First, encryption related operations take up more time in larger models, leaving more potential space for BatchCrypt. Second, since layers are batched separately, larger layers have higher chances forming long batches. BatchCrypt's speedup can be up to two orders of magnitude, which easily offset the extra epochs needed for convergence caused by quantization (§6.2).

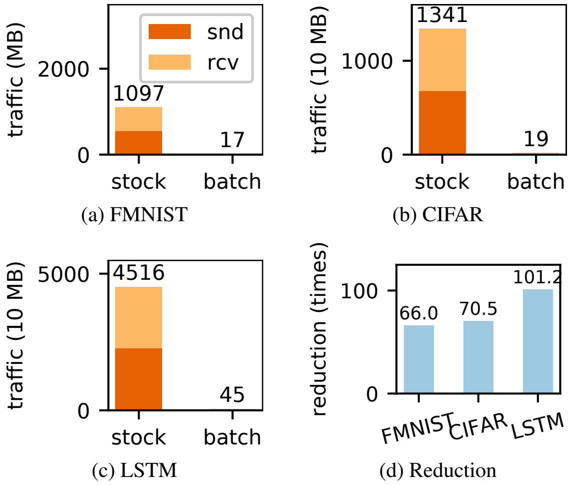
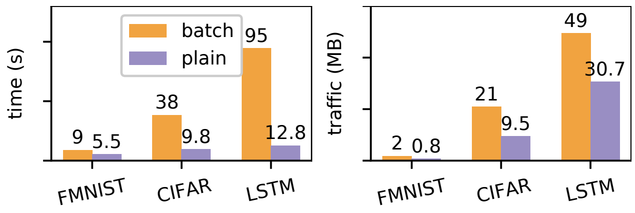


Figure 8: Comparison of the network traffic incurred in one training iteration using the stock FATE implementation and BatchCrypt.

BatchCrypt vs. Plaintext Learning We next compare BatchCrypt with the plain distributed learning where no encryption is involved—an ideal baseline that offers the optimal performance. Fig. 9 depicts the iteration time and the network

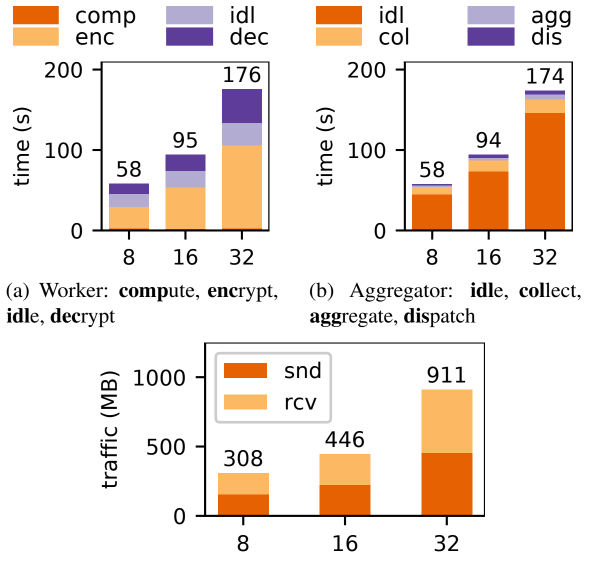


(a) Time (b) Communication

Figure 9: Time and communication comparisons of one iteration on workers between BatchCrypt and plain distributed learning without encryption.

Table 4: Projected total training time and network traffic usage until convergence for the three models. The converged test accuracy for FMNIST, CIFAR as well as loss for LSTM and their corresponding epoch numbers are listed in the table.

Model	**Mode**	**Epochs**	**Acc./Loss**	**Time** (h)	**Traffic** (GB)
FMNIST	stock	40	88.62%	122.5	2228.3
	batch	68	88.37%	8.9	58.7
	plain	40	88.62%	3.2	11.17
CIFAR	stock	285	73.97%	9495.6	16422.0
	batch	279	74.04%	131.3	227.8
	plain	285	73.97%	34.2	11.39
LSTM	stock	20	0.0357	8484.4	15347.3
	batch	23	0.0335	105.2	175.9
	plain	20	0.0357	12.3	10.4



(a) Worker: **comp**ute, **enc**rypt, **idl**e, **dec**rypt

(b) Aggregator: **idl**e, **col**lect, **agg**regate, **dis**patch

(c) Aggregator traffic

Figure 10: Breakdown of iteration time and communication traffic of BatchCrypt with LSTM model with various quantization bit widths in one iteration. The corresponding batch sizes for bit width 8, 16, and 32 are 200, 100, and 50, respectively.

footprint under the two implementations. While encryption remains the major bottleneck, BatchCrypt successfully reduces the overhead by an order of magnitude, making it practical to achieve the same training results as the plain distributed setting. Note that encrypted numbers in FATE each carries redundant information such as public keys, thus causing the communication inflation compared with the plain version. Such inflation can be reduced if FATE employs some optimized implementation.

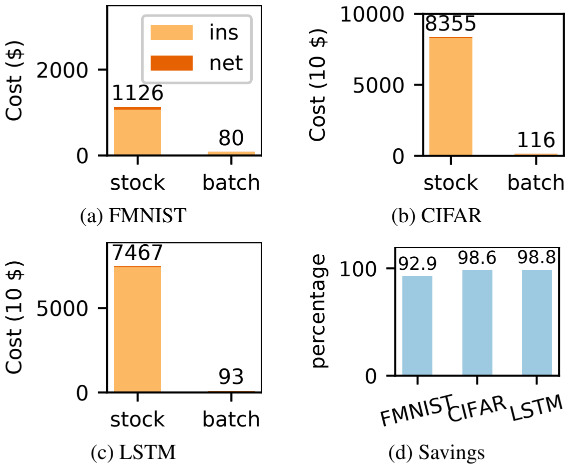
Training to Convergence Our previous studies mainly focus on a single iteration. Compared with stock FATE and plain distributed learning, BatchCrypt requires a different number of iterations to converge. We hence evaluate their end-to-end performance by training ML models till convergence. As this would take exceedingly long time and high cost if performed in real deployment, we instead utilize our simulation in §6.2 and iteration profiling results to project the total time and network traffic needed for convergence.

Table 4 lists our projection results of the three solutions. Compared with the stock implementation in FATE, BatchCrypt dramatically reduces the training time towards convergence by 13.76×, 72.32×, and 80.65× for FMNIST, CIFAR, and LSTM, respectively. In the meantime, the network footprints shrink by 37.96×, 72.01×, 87.23×, respectively. We stress that these performance improvements are achieved without degrading the trained model quality. On the other hand, BatchCrypt only slows down the overall training time by 1.78×, 2.84×, and 7.55× for the three models compared with plain learning—which requires no encryption and hence achieves the fastest possible training convergence. In summary, BatchCrypt significantly reduces both the computation and communication overhead caused by HE, enabling efficient HE for cross-silo FL in production environments.

6.4 Batching Efficiency

We have shown in §6.2 that ML applications have different levels of sensitivity towards gradient quantization. It is hence essential that BatchCrypt can efficiently batch quantized values irrespective of the chosen quantization bit width. Given an HE key, the longest plaintext it can encrypt is determined by the key size, so the shorter the quantization width is, the larger the batch size is, and the higher the potential speedup could be. We therefore look into how our BatchCrypt implementation can exploit such batching speedup.

We evaluate BatchCrypt by varying the batch size. In particular, we train the LSTM model on the geo-distributed clients with different quantization widths 8, 16, and 32. The corresponding batch sizes are set respectively to 200, 100, and 50. We ran the experiments for 50 iterations, and illustrate the average statistics in Fig. 10. Figs. 10a and 10b show the time breakdown in the three experiments. It is clear that employing a shorter quantization bit width enables a larger batch size, thus leading to a shorter training time. Note that the speedup going from 8-bit to 16-bit is smaller compared with that from 16-bit to 32-bit, because HE operations become less of a bottleneck with larger batch size. Fig. 10c depicts the



(a) FMNIST (b) CIFAR (c) LSTM (d) Savings

Figure 11: Total cost until convergence between FATE's stock implementation and BatchCrypt, **ins**tance and **net**work costs are highlighted separately.

accumulated network traffic incurred in one iteration, which follows a similar trend as that of the iteration time. In conclusion, BatchCrypt can efficiently exploit batching thanks to its optimized quantization. Similar to [26, 51], BatchCrypt's batching scheme reduces both the computation and communication cost linearly as the batch size increases. In fact, if lattice-based HE algorithms are adopted, one can replace BatchCrypt's batching scheme with that of [26, 51], and still benefit from BatchCrypt's accuracy-preserving quantization.

6.5 Cost Benefits

The reduced computation and communication overheads enable significant cost savings: sustained high CPU usage leads to high power consumption, while ISPs charge for bulk data transfer over the Internet. As our evaluations were conducted in EC2, which provides a runtime environment similar to the organization's own datacenters, we perform cost analysis under the AWS pricing scheme. The hourly rate of our cluster is $8.758, while the network is charged based on outbound traffic for $0.042, $0.050, $0.042, $0.048, $0.055 per GB for the regions listed in Table 2.

We calculate the total cost for training until convergence in Table 4 and depict the results in Fig. 11. As both computation and communication are reduced substantially, BatchCrypt achieves huge cost savings over FATE. While the instance cost reduction is the same as the overall speedup in Table 4, BatchCrypt lowers the network cost by 97.4%, 98.6% and 98.8% for FMNIST, CIFAR, and LSTM, respectively.

7 Discussion

Local-update SGD & Model Averaging Local-update SGD & model averaging is another common approach to reducing the communication overhead for FL [22, 40], where the aggregator collects and averages model weights before propagating them back to clients. Since there are only addition operations involved, BatchCrypt can be easily adopted.

Split Model Inference In many FL scenarios with restrictive privacy requirement, a trained model is split across clients, and model inference involves coordination of all those clients [23, 61]. BatchCrypt can be used to accelerate the encryption and transmission of the intermediate inference results.

Flexible Synchronization There have been many efforts in amortizing the communication overhead in distributed SGD by removing the synchronization barriers [24, 34, 62]. Although we only evaluate BatchCrypt's performance in synchronous SGD, our design allows it to take advantage of the flexible synchronization schemes proposed in the literature. This is not possible with Secure Aggregation [9].

Potential on Large Models Recent research and our evaluations show that more sophisticated ML models are more resilient to quantization noise. In fact, certain models are able to converge even with 1- or 2-bit quantization [8, 58]. The phenomenon promises remarkable improvement with BatchCrypt, which we will explore in our future work.

Applicability in Vertical FL Vertical FL requires complicated operations like multiplying ciphertext matrices [38, 61]. Batching over such computation is beyond BatchCrypt's current capability. We will leave it as a future work.

8 Concluding Remark

In this paper, we have systematically studied utilizing HE to implement secure cross-silo FL. We have shown that HE related operations create severe bottlenecks on computation and communication. To address this problem, we have presented BatchCrypt, a system solution that judiciously quantizes gradients, encodes a batch of them into long integers, and performs batch encryption to dramatically reduce the encryption overhead and the total volume of ciphertext. We have implemented BatchCrypt in FATE and evaluated its performance with popular machine learning models across geo-distributed datacenters. Compared with the stock FATE, BatchCrypt accelerates the training convergence by up to 81× and reduces the overall traffic by 101×, saving up to 99% cost when deployed in cloud environments.

Acknowledgement

We thank our shepherd, Brandon Lucia, and the anonymous reviewers for their valuable feedbacks that help improve the quality of this work. This work was supported in part by RGC ECS grant 26213818, WeBank-HKUST research collaboration grant 2019, NSF CCF-1756013 and NSF IIS-1838024. Chengliang Zhang was supported by the Hong Kong PhD Fellowship Scheme.

References

[1] Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NeurIPS* (2017).

[2] Anderson, A. G., and Berg, C. P. The high-dimensional geometry of binary neural networks. In *ICLR* (2018).

[3] Aws deep learning ami. https://aws.amazon.com/machine-learning/amis/, 2019.

[4] Banner, R., Hubara, I., Hoffer, E., and Soudry, D. Scalable methods for 8-bit training of neural networks. In *NeurIPS* (2018).

[5] Banner, R., Nahshan, Y., and Soudry, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *NeurIPS* (2019).

[6] Barker, E., Barker, W., Burr, W., Polk, W., and Smid, M. Recommendation for key management part 1: General (revision 3). *NIST special publication 800*, 57 (2012), 1–147.

[7] Baskin, C., Schwartz, E., Zheltonozhskii, E., Liss, N., Giryes, R., Bronstein, A. M., and Mendelson, A. Uniq: Uniform noise injection for non-uniform quantization of neural networks. *arXiv preprint arXiv:1804.10969* (2018).

[8] Bernstein, J., Wang, Y.-X., Azizzadenesheli, K., and Anandkumar, A. signsgd: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434* (2018).

[9] Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 1175–1191.

[10] Bottou, L., and Bousquet, O. The tradeoffs of large scale learning. In *NeurIPS* (2008).

[11] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT) 6*, 3 (2014), 1–36.

[12] California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa, 2018.

[13] Cheng, K., Fan, T., Jin, Y., Liu, Y., Chen, T., and Yang, Q. Secureboost: A lossless federated learning framework. *arXiv preprint arXiv:1901.08755* (2019).

[14] Courbariaux, M., Bengio, Y., and David, J.-P. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[15] Data61, C. Python paillier library. https://github.com/data61/python-paillier, 2013.

[16] Du, W., Han, Y. S., and Chen, S. Privacy-preserving multivariate statistical analysis: Linear regression and classification. In *Proceedings of the 2004 SIAM international conference on data mining* (2004), SIAM, pp. 222–233.

[17] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). https://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016.

[18] FATE (Federated AI Technology Enabler). https://github.com/FederatedAI/FATE, 2019.

[19] Ge, T., and Zdonik, S. Answering aggregation queries in a secure system model. In *VLDB* (2007).

[20] Tensorflow Federated. https://www.tensorflow.org/federated, 2019.

[21] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision. In *ICML* (2015).

[22] Haddadpour, F., Kamani, M. M., Mahdavi, M., and Cadambe, V. Local sgd with periodic averaging: Tighter analysis and adaptive synchronization. In *NeurIPS* (2019).

[23] Hardy, S., Henecka, W., Ivey-Law, H., Nock, R., Patrini, G., Smith, G., and Thorne, B. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv preprint arXiv:1711.10677* (2017).

[24] Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., Gibson, G. A., Ganger, G., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *NeurIPS* (2013).

[25] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[26] Juvekar, C., Vaikuntanathan, V., and Chandrakasan, A. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 1651–1669.

[27] Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z., Cormode, G., Cummings, R., et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977* (2019).

[28] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[29] Koloskova, A., Stich, S. U., and Jaggi, M. Decentralized stochastic optimization and gossip algorithms with compressed communication. In *ICML* (2019).

[30] Konečný, J., McMahan, H. B., Ramage, D., and Richtárik, P. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527* (2016).

[31] Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. Tech. rep., Citeseer, 2009.

[32] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *NeurIPS* (2012).

[33] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014), USENIX.

[34] Lian, X., Huang, Y., Li, Y., and Liu, J. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NeurIPS* (2015).

[35] Lin, T., Stich, S. U., Patel, K. K., and Jaggi, M. Don't use large mini-batches, use local sgd. *arXiv preprint arXiv:1808.07217* (2018).

[36] Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[37] Liu, C., Chakraborty, S., and Verma, D. Secure model fusion for distributed learning using partial homomorphic encryption. In *Policy-Based Autonomic Data Governance*. Springer, 2019, pp. 154–179.

[38] Liu, Y., Chen, T., and Yang, Q. Secure federated transfer learning. *arXiv preprint arXiv:1812.03337* (2018).

[39] McMahan, H. B., Moore, E., Ramage, D., Hampson, S., et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629* (2016).

[40] McMahan, H. B., Moore, E., Ramage, D., and y Arcas, B. A. Federated learning of deep networks using model averaging. *ArXiv abs/1602.05629* (2016).

[41] Migacz, S. 8-bit inference with tensorrt. In *GPU technology conference* (2017), vol. 2, p. 7.

[42] Mohassel, P., and Rindal, P. Aby 3: a mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 35–52.

[43] Mohassel, P., and Zhang, Y. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 19–38.

[44] Nikolaenko, V., Weinsberg, U., Ioannidis, S., Joye, M., Boneh, D., and Taft, N. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 334–348.

[45] Cybersecurity Law of the People's Republic of China. http://www.lawinfochina.com/display.aspx?id=22826&lib=law, 2017.

[46] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques* (1999), Springer, pp. 223–238.

[47] Pathak, M., Rane, S., and Raj, B. Multiparty differential privacy via aggregation of locally trained classifiers. In *NeurIPS* (2010).

[48] Phong, L. T., Aono, Y., Hayashi, T., Wang, L., and Moriai, S. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security 13*, 5 (2018), 1333–1345.

[49] Ryffel, T., Trask, A., Dahl, M., Wagner, B., Mancuso, J., Rueckert, D., and Passerat-Palmbach, J. A generic framework for privacy preserving deep learning. *arXiv preprint arXiv:1811.04017* (2018).

[50] San, I., At, N., Yakut, I., and Polat, H. Efficient paillier cryptoprocessor for privacy-preserving data mining. *Security and communication networks 9*, 11 (2016), 1535–1546.

[51] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, Apr. 2020. Microsoft Research, Redmond, WA.

[52] Shokri, R., and Shmatikov, V. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (2015), ACM, pp. 1310–1321.

[53] Soudry, D., Hubara, I., and Meir, R. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NeurIPS* (2014).

[54] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[55] Text generation with an rnn. https://www.tensorflow.org/tutorials/text/text_generation, 2019.

[56] Wang, J., and Joshi, G. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv preprint arXiv:1810.08313* (2018).

[57] WeBank. https://www.webank.com/en/, 2019.

[58] Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *NeurIPS* (2017).

[59] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. The marginal value of adaptive gradient methods in machine learning. In *NeurIPS* (2017), pp. 4148–4158.

[60] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[61] Yang, Q., Liu, Y., Chen, T., and Tong, Y. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST) 10*, 2 (2019), 12.

[62] Zhang, C., Tian, H., Wang, W., and Yan, F. Stay fresh: Speculative synchronization for fast distributed machine learning. In *ICDCS* (2018), IEEE.

[63] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).