SparkFHE: Distributed Dataflow Framework with Fully Homomorphic Encryption

Peizhao Hu[1], **Asma Aloufi**[1], **Adam Caulfield**[1], **Kim Laine**[2], **and Kristin Lauter**[2]
[1] Rochester Institute of Technology, USA, [2] Microsoft Research, Redmond, USA
Corresponding email: Peizhao.Hu@rit.edu

Abstract

We propose a new framework, which aims to enable large-scale privacy-preserving machine learning (PPML) in the cloud. In this extended abstract, we discuss the integration of Apache Spark and fully homomorphic encryption (FHE), thus the name SparkFHE. The SparkFHE framework enables Spark to perform computations on encrypted data without requiring the secret key hence preserving user data privacy, and scales up homomorphic algorithms for larger datasets through efficient cluster programming. We present the architecture design, programming abstractions, and mappings of applications such as private set intersection and logistic regression, into the dataflow model of Spark. Furthermore, we discuss preliminary results to validate our designs.

1 Introduction

Cloud computing is becoming indispensable for building machine learning pipelines at scale. At the core of cloud computing is distributed data processing frameworks partitioning large volume of user data and orchestrating task allocation to achieve parallelization in a single-program-multiple-data (SPMD) manner. Recent data processing frameworks, such as Apache Spark [1], Apache Flink [2], and others [3, 4, 5], adopt the distributed dataflow model to maximize in-memory processing and optimize task allocation, reducing significant data movement overhead and data processing time.

However, many ML tasks require sensitive user data, for example genome-wide association study (GWAS), and support for PPML on the current cloud infrastructures is lacking. We propose a new framework — SparkFHE, to address the needs of PPML in the cloud. SparkFHE integrates fully homomorphic encryption (FHE) with Spark so that Spark can perform computations on encrypted data without requiring the secret keys. This integration brings new capability to Spark users who might want to develop PPML algorithms and benefit from the highly efficient data processing environment. In addition to our contribution in the architecture design, we discuss the programming abstractions for easing the development PPML algorithms in the Spark context. To validate our designs, we present case-study of mapping two example applications into the dataflow model using Spark's resilient distributed datasets (RDD) [6], and discuss some preliminary evaluation results.

2 System Architecture and Programming Abstractions

The two main goals when designing SparkFHE are to enable Spark to perform homomorphic computations on encrypted data, and to leverage the efficiency of parallelized data processing of Spark to scale homomorphic computations for large datasets. While achieving these two goals, we develop programming abstraction to ease the development of PPML algorithms for Spark users who are cryptography novice. Figure 1 illustrates the overall architecture of SparkFHE and its components for FHE support, parameter setup, experimentation, and a validation setup using off-the-shelf cloud computing software. The core module is libSparkFHE, which is a C++ shared library. This module defines data types that incorporate various FHE libraries such as SEAL [7] and HElib [8] into SparkFHE and implements primitive functions and homomorphic algorithms to support our programming abstraction. Functionalities provided by this module is exposed to the Spark users

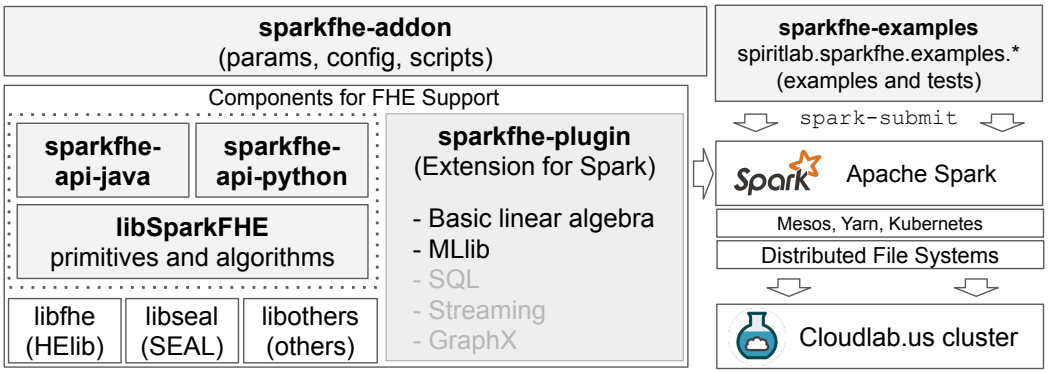


Figure 1: SparkFHE system architecture (proof-of-concept demo, https://bit.ly/2SbxhtJ).

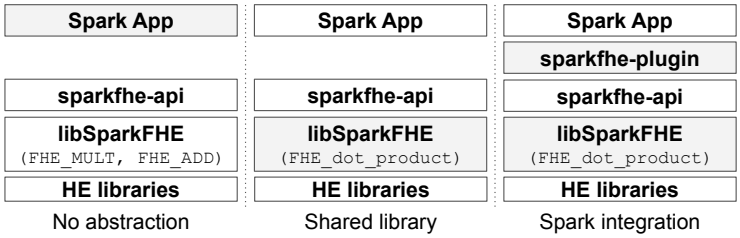


Figure 2: Level of abstraction (optimization in shaded boxes).

through two API modules for standard Spark and PySpark. We have developed our programming abstraction as a third-party plugin for Spark. Our current prototype supports basic linear algebra over vectors or matrices of ciphertexts in either batching or non-batching mode. Using these basic linear algebra operations, we have developed simple ML algorithms such as logistic regression. We have developed a prototype of SparkFHE based on this system architecture, integrated with the latest version of Spark core, and built an experimentation pipeline using Mesos for resource management and Hadoop as a distributed filesystem. We have tested our deployment of the SparkFHE framework on Cloudlab.us cluster.

One of the main contributions of SparkFHE is the new programming abstraction. The programming abstraction hides the complexity of the underlying homomorphic operations. We have explored the feasibility and efficiency of programming abstraction at three different levels, as shown in Figure 2. When there is no abstraction, our library and API expose homomorphic primitive operations such as addition and multiplication to Spark users. Spark users have to develop their PPML algorithms using a combination of small lambda functions, as illustrated in Listing 1. The biggest concern is the overhead of transferring function calls between programming languages. Every call to the homomorphic primitive operations requires transferring the call as well as data to the C++ shared library and then to the corresponding FHE library such as SEAL or HElib. This transfer overhead is substantial, as observed in our experimental results (see Supplement) that compare the performance of different programming abstractions. Of course we can develop a function in our C++ library, thus the transferring of function calls is closer to the FHE libraries. Although this approach is straightforward, this design does not fully leverage the functionalities provided by Spark; for example, the abstractions over RDD, dataframe, and dataset. As a result, these two partial integration methods fail to fully leverage Spark to parallelize homomorphic computations over large ciphertexts. To reduce the transferring overhead and ease the development for Spark users, we have developed a plugin module (see Figure 1) that extends the capability of Spark and fully integrates FHE functionalities provided by our library and the underlying FHE libraries into Spark. As a result of this integration, the task of computing dot-product of two vectors of ciphertexts can be rewritten using the transformer design pattern. This design pattern is also used by many builtin ML algorithms in the Spark's MLlib module.

3 Modeling of Algorithms as Dataflow Graphs

Another important aspect of fully integrating Spark with FHE is to leverage Spark's efficient cluster computing to optimize homomorphic computations, thus reduce the evaluation time. This idea goes beyond simply parallelizing Spark jobs on multiple computing nodes. In Spark, the dataflow

Listing 1: Calculating dot-product from two vectors of ciphertexts without abstraction.

```
JavaRDD<Ciphertext> product_rdd = ctxts_rdd.map(tuple -> {
            return SparkFHE.getInstance().fhe_multiply(tuple._1(), tuple._2());
        });
Ciphertext result = product_rdd.reduce((x, y) -> {
            return SparkFHE.getInstance().fhe_add(x, y);
        });
```

Listing 2: Transforming a vector of ciphertexts using an integrated function (dot-product).

```
JavaRDD<CtxtVector> data = jsc.parallelize(Arrays.asList(
    CtxtVectors.dense(ctxt1, ctxt2, ctxt3)
));
CtxtVector transformationVector = CtxtVectors.dense(ctxtA, ctxtB, ctxtC);
DotProduct dp = new DotProduct(transformationVector);
JavaRDD<CtxtVector> result = dp.transform(data);
```

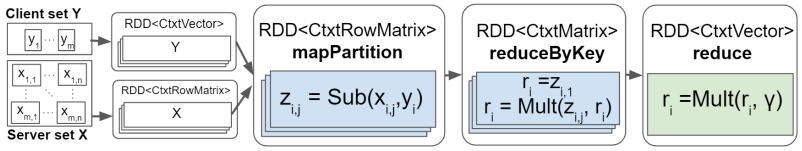
model in the form of a directed acyclic graph (DAG) provides important metadata for a scheduler to optimize task allocation. For example, Spark uses this DAG to identify tasks that can be combined together in its staging process. Tasks bundled together in a stage are executed on a worker fully utilizing in-memory processing, hence avoiding data movement overhead. As an integral part of the development of SparkFHE, we have explored the feasibility of mapping algorithms into dataflow model. Here, we present two example FHE algorithms: private set intersection and private evaluation of logistic regression.

Private set intersection (PSI) [9] is an algorithm for securely determining a common subset between two datasets without revealing other information. Through SparkFHE, Spark users are able to represent this algorithm as a dataflow graph that can be directly implemented as a sequence of MapReduce functions in the RDD. Figures 3a and 3b illustrate the pseudocode and dataflow diagram for PSI. In order to implement this, first the the dataset X and the inquiry set Y are encrypted and stored in RDD objects containing `CtxtRowMatrix` and `CtxtVector` respectively. We have developed prototype for these new datatypes for ciphertexts based on their plaintext version, `RowMatrix` and `Vector` respectively, to support the mapping. Using the `CtxtRowMatrix` object in RDD enables Spark users to parallelize element-wise operations by creating row-wise partitions. After that a `mapPartition` is used to homomorphically subtract Y from X through element-wise subtraction of each element y_i of Y from each element in row x_i of X. Reduction is then completed to reduce the previous result to a vector representing the result for each y_i in Y by multiplying the result element-wise. In the final reduce step, each element in the final `CtxtVector` is multiplied by a random number to mask unmatched results.

The second example demonstrates logistic regression [10], a common machine learning model for predicting class association based on a set of data. Figures 3c and 3d demonstratre the pseudocode and dataflow diagram for logistic regression. Logistic regression requires multiple iterations to complete model training. Thus, mapping the dataflow diagram of logistic regression to the MapReduce functions in RDD demonstrates SparkFHE support for iterative algorithms. Since we are dealing with encrypted input data, we model this algorithm using our new datatypes for ciphertexts. The training set X is stored in a RDD of `CtxRowMatrix`, and the classification vector Y is mapped to a RDD storing `CtxtVector`. The weights are initialized to an encryption of zero and passed as an argument. The first `mapPartition` is used to compute the dot product of the rows of X by the weights with intermediate result u. After this in logistic regression the result is typically evaluated with the sigmoid function to generate an associated probability. However, the sigmoid function cannot be represented primitive algebraic functions and therefore cannot be executed homomorphically. Because of this, we use the following polynomial approximation of the sigmoid from [10]: $0.5 + 0.197x^2$. This polynomial approximation is represented in the function `evalPoly`. To execute `evalPoly` on the previous result u, u is stored in RDD of `CtxtVector`, and a `mapPartition` is used to execute `evalPoly` the RDD partitions of u. After that is completed, an intermediate result is stored in v. Another `mapPartition` is used to homomorphically subtract the previous result v from Y to realize the accuracy v' of the current weights. After this gradient descent should be performed. This includes using a `mapPartition` compute the dot product of X and v'. then the result is multiplied with the columns of the `CtxRowMatrix` to generate the gradient. Then the gradient values are added to each of the weights to complete one iteration. This process is repeated for a number of iterations set by the user.

```
for i in range(1, m) do
  for j in range(1, n) do
    Z[i][j] ← FHE.Subtract(X[i][j], Y[i])
  end for
end for
for i in range(1, m) do
  R[i] ← Z[i][1]
  for j in range(2, n) do
    R[i] ← FHE.Multiply(Z[i][j], R[i])
  end for
end for
for i in range(1, m) do
  R[i]← FHE.Multiply(R[i], γ)
end for
```

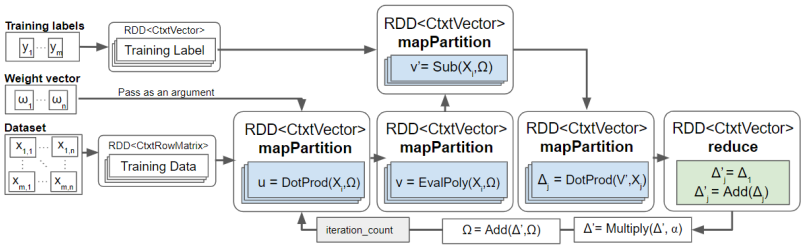
(a) PSI



(b) Private Set Intersection

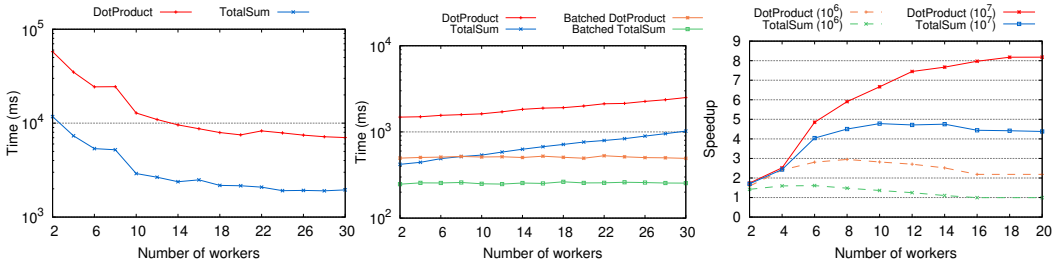
```
for iter in range(1, iterations) do
  for i in range(1, m) do
    U[i] ← FHE.DotProd(X[i], Ω)
    V[i] ← FHE.EvalPoly(U[i])
    V`[i] ← FHE.Subtract(V[i], Y[i])
  end for
  for j in range(1, n) do
    Δ[j] ← FHE.DotProd(V`, X[j])
    Δ`[j] ← FHE.Multiply(Δ[j], α)
    Ω[j] ← FHE.Add(Δ`[j],Ω[j])
  end for
end for
```

(c) Logistic Regression



(d) Logistic Regression

Figure 3: Pseudocode and mappings to the dataflow model.



(a) Strong scaling (b) Weak scaling (c) Parallel Speedup

Figure 4: Speedup in SparkFHE as measured by strong and weak scaling.

4 Validation and Discussions

To validate our proposed framework, we constructed a proof-of-concept prototype, deployed our SparkFHE framework on Cloudlab.us [11], and conducted initial experimental evaluations on systems with Intel 10-core Xeon E5-2660, 2.60GHz, 164GB RAM, and Ubuntu 18.04.2. We investigated the performance gain of Spark for HE algorithms in a controlled environment, avoiding network fluctuation. Figure 4 shows the preliminary results of performance gain by SparkFHE based on two well-known metrics in parallel computing, strong- and weak-scaling. As shown in Fig. 4b, as the problem size grows proportionally with the number of workers, such that each worker node processes 100 encrypted values, we observe an approximately fixed running time; hence, achieving an acceptable weak-scaling. Figure 4a shows that for a fixed problem size of 10^4 encrypted values, we observe a speedup as the number of workers increases, therefore achieving good strong-scaling. Note, no speedup was observed for the batching version of this problem because encrypting 10^4 data yielded two batched ciphertexts. On the other hand, Fig. 4c illustrates the growing speedup for batched ciphertexts as the problem size scales up to 10^6 and 10^7 encrypted values.

5 References

[1] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. DOI: 10.1145/2934664.

[2] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38 (2015), pp. 28–38.

[3] Tyler Akidau et al. "MillWheel: fault-tolerant stream processing at internet scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[4] Derek G Murray et al. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 439–455.

[5] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: (2015).

[6] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28.

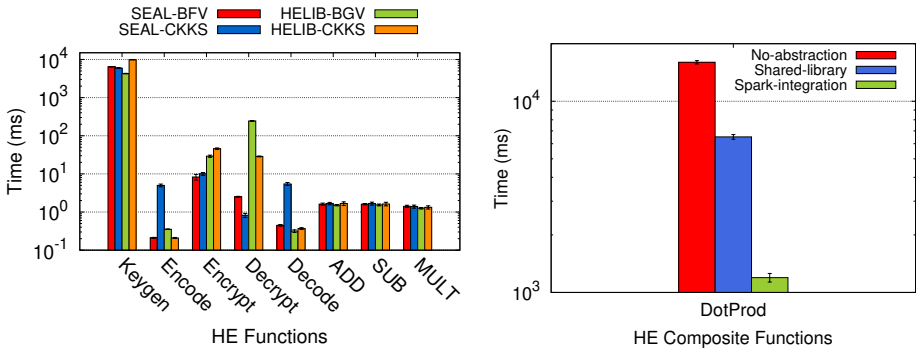
[7] Kim Laine et al. "Simple Encrypted Arithmetic Library v3.2.0". In: *Technical Report*. 2018.

[8] Shai Halevi and Victor Shoup. *An Implementation of homomorphic encryption*. 2013. URL: https://github.com/homenc/HElib/.

[9] Hao Chen, Kim Laine, and Peter Rindal. "Fast Private Set Intersection from Homomorphic Encryption". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: ACM, 2017, pp. 1243–1255. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134061. URL: http://doi.acm.org/10.1145/3133956.3134061.

[10] Andrey Kim et al. "Logistic regression model training based on the approximate homomorphic encryption". In: *BMC medical genomics* 11.4 (2018), p. 83.

[11] Dmitry Duplyakin et al. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.



(a) Different HE schemes (b) Different levels of abstraction

Figure 5: Performance comparison of different schemes and levels of abstraction.

Supplement: Performance of programming abstractions

Our proposed SparkFHE framework provides support for computation with different HE libraries and schemes. Here, we analyze the performance of basic HE functions (e.g., `KeyGen`, `Encrypt`, `Decrypt`, `Encode`, `Decode`) and primitive HE arithmetic algorithms (Addition, Subtraction, Multiplication). Figure 5a shows the running times for four cases with supported libraries (Microsoft SEAL and HElib) and schemes (BFV, CKKS, BGV).

In all four cases, the key generation step is performed only once and takes around 1 second to generate a key pair. The running time also includes the internal abstraction layer functions to store the keys on the client's side. The following four HE functions are also performed once, for each ciphertext, on the client's side before uploading the encrypted data to the cloud for evaluation. On the other hand, the HE primitive arithmetic algorithms take place on the cloud side on two ciphertexts. Each one of them takes around 1 millisecond to be performed in parallel within Spark.

In Fig. 5b, we inspect the data transfer overhead when using different levels of programming abstractions (Figure 2) in the SparkFHE framework. We measure the running time for homomorphically evaluating the Dot Product function on 10^3 individually encrypted values with the SEAL-BFV scheme in our framework. We observe that using function from the shared library offers approximately $\times 4$ speedup compared with no abstraction; that is, writing the dot-product function as separate map and reduce lambda functions (see Listing 1).