SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning

Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang, *Ant Group*

https://www.usenix.org/conference/atc23/presentation/ma

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning

Junming Ma, Yancheng Zheng,* Jun Feng, Derun Zhao, Haoqi Wu
Wenjing Fang, Jin Tan,† Chaofan Yu, Benyu Zhang, Lei Wang

Ant Group

Abstract

With the increasing public attention to data security and privacy protection, privacy-preserving machine learning (PPML) has become a research hotspot in recent years. Secure multi-party computation (MPC) that allows multiple parties to jointly compute a function without leaking sensitive data provides a feasible solution to PPML. However, developing efficient PPML programs with MPC techniques is a great challenge for users without cryptography backgrounds.

Existing solutions require users to make efforts to port machine learning (ML) programs by mechanically replacing APIs with PPML versions or rewriting the entire program. Different from the existing works, we propose SecretFlow-SPU, a performant and user-friendly PPML framework compatible with existing ML programs. SecretFlow-SPU consists of a frontend compiler and a backend runtime. The frontend compiler accepts an ML program as input and converts it into an MPC-specific intermediate representation. After a series of delicate code optimizations, programs will be executed by a performant backend runtime as MPC protocols. Based on SecretFlow-SPU, we can run ML programs of different frameworks with minor modifications in a privacy-preserving manner.

We evaluate SecretFlow-SPU with state-of-the-art MPC-enabled PPML frameworks on a series of ML training tasks. SecretFlow-SPU outperforms these works for almost all experimental settings (23 out of 24). Especially under the wide area network, SecretFlow-SPU is up to 4.1× faster than MP-SPDZ and up to 2.3× faster than TF Encrypted.

1 Introduction

Privacy-preserving machine learning (PPML) [24, 27, 34, 43, 44, 47, 49, 56, 57] has been gaining popularity due to the pervasive usage of machine learning (ML) and attendant privacy problems. Secure multi-party computation (MPC) [39], a cryptographic technique that enables multiple parties to jointly compute a function without leaking each party's private inputs, brings a provable and practical solution to ML users with strong privacy concerns. For example, financial and medical data analysts can collaboratively train a model on private datasets that contain sensitive information.

However, incorporating MPC techniques in ML applications introduces great challenges due to the natural differences between these two fields. MPC experts mainly focus on designing performant cryptographic protocols for low-level computation primitives. In contrast, ML practitioners are more accustomed to constructing high-level ML models using user-friendly frameworks that encapsulates commonly-used ML building blocks. Consequently, it poses a massive obstacle for ML users without cryptography expertise to achieve complex PPML tasks efficiently in real-world scenarios.

A series of works have been proposed to eliminate this obstacle. EzPc [9], ABY [15], MP-SPDZ [29], etc. [18] design domain-specific languages (or use high-level languages) to provide general purpose MPC compilers and support arbitrary computations upon MPC. These works significantly reduce the difficulty of developing MPC programs and allow for MPC-specific compilation optimizations. Whereas, these works remain a significant gap from mainstream ML frameworks on API designs, thus lacking user-friendliness to develop complex ML programs.

TF Encrypted [14] and CrypTen [33] take a step further in this direction by providing general ML interfaces with MPC implementations. These works mimic the existing ML frameworks' API designs (e.g., TensorFlow [5] and PyTorch [45]) to hide the underlying MPC cryptographic details and gain further user-friendliness. However, efforts still need to be made to port ML programs from TensorFlow/PyTorch by substituting PPML version APIs mechanically. Take CrypTen as an example: given a pre-defined PyTorch model, the user must manually re-write the model training/prediction programs by replacing PyTorch tensors, loss function, and optimizer with CrypTen counterparts. Besides, these frameworks rely on TensorFlow or PyTorch as their underlying runtime, which lacks

*Junming and Yancheng contribute equally in this work.
†Corresponding author: tanjin.tj@antgroup.com.

MPC domain-specific knowledge for compilation optimizations.

A question then arises: *can we efficiently run ML programs of mainstream frameworks in a privacy-preserving manner?* As an attempt to answer this question, we propose SecretFlow-SPU in this paper. For simplicity, we will refer to it as SPU throughout the rest of this paper. SPU is a general-purpose PPML framework designed to bridge the gap between ML and MPC communities more naturally. The core components of SPU include a frontend compiler and a backend MPC runtime. SPU provides users with Python APIs to accept an ML program (with minor modifications to specify protected data) as input, and the frontend compiler emits a customized intermediate representation (IR) named PPHLO (short for **P**rivacy-**P**reserving **H**igh-**L**evel **O**perations) as output. The backend runtime is a virtual device built on multiple connected computing nodes, which receives PPHLO and executes it as MPC protocol implementations among nodes to complete private ML training or prediction.

SPU's architecture makes it friendly to both ML and MPC developers. On the one hand, ML developers can conveniently run ML applications developed through mainstream ML frameworks in a privacy-preserving manner on SPU (Section 3.3) without the cryptographic knowledge of MPC. Besides, SPU is not bound to one specific ML framework. Diverse frameworks and libraries can be supported in SPU if there is a path from ML source code to PPHLO. On the other hand, SPU provides great extensibility to MPC protocol developers, who only need to focus on designing fundamental MPC primitives and implementing corresponding APIs defined by SPU. New MPC protocol supports can be easily supplemented without caring about high-level ML workflows (Section 3.6.1).

Besides user-friendliness, PPHLO enables us to propose and implement MPC-specific optimizations at both frontend and backend to achieve high performance. At the frontend, we observe that traditional ML frameworks usually generate a computation graph that is not optimal for MPC computations. The reason behind the observation is that MPC computations have a rather different cost model than plaintext computations due to additional communication overhead. Based on PPHLO, we design and implement several compiler passes to generate more efficient IR (Section 3.5). At the backend, we implement strategies such as vectorization and streaming to reduce MPC communication overhead. Meanwhile, SPU backend runtime employs inter- and intra-operation concurrency to execute PPHLO efficiently (Section 3.6.2).

We develop SPU frontend and backend in C++ and provide PPML developers with Python APIs to run applications. We mainly use ML programs written in JAX [8] to evaluate SPU's performance and user-friendliness. For performance, we use three state-of-the-art MPC-enabled PPML frameworks (i.e., MP-SPDZ [29], TF Encrypted [14], and CrypTen [33]) as the baseline. We train four common-evaluated neural networks on the MNIST [37] dataset for image classification under both local area network (LAN) and wide area network (WAN) settings. SPU achieves comparable classification accuracy and superior training speed. Concretely, SPU outperforms the state-of-the-art works for almost all the settings (23 out of 24). Especially under the WAN setting, SPU is up to 4.1× faster than MP-SPDZ and up to 2.3× faster than TF Encrypted.

Regarding user-friendliness, we evaluate SPU by running JAX programs from popular open-source JAX projects' official examples. We only need to modify a few lines of code to make these examples run seamlessly on SPU. Experimental results show that SPU can be easily applied to other models such as Long Short-Term Memory [23] and Variational Auto-Encoder [32]. This compatibility is hard to achieve for existing MPC-enabled frameworks. Moreover, we also validate SPU's feasibility in supporting different ML frameworks by running TensorFlow and PyTorch programs.

The contributions we make in this paper are summarized as follows:

- We design and implement SPU as the first MPC-enabled PPML framework to support ML programs (with minor modifications) from different mainstream ML frameworks, significantly accelerating the development, testing, debugging, and deployment of PPML applications.
- We design an MPC-specific IR, i.e., PPHLO, which connects ML and MPC worlds. Besides, we propose/implement a series of compilation optimizations and develop a high-performance runtime to execute PPHLO.
- We validate SPU with a series of experiments on performance and user-friendliness. The experimental results demonstrate SPU's efficiency and ease of use.
- We open-source SPU to bolster the advancement of PPML for academic and industry communities. The code is available at https://github.com/secretflow/spu.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the background of ML compilers and MPC. We describe the design details of SPU in Section 3. Section 4 describes the system implementation and evaluation results. Section 5 describes SPU's limitations. Related work is discussed in Section 6. Finally, we conclude this paper in Section 7.

2 Background

SPU is an interdisciplinary work of ML compiler and MPC. In order to better understand the motivation and design of SPU, we give a more detailed description of the background of ML compiler and MPC in this section.

2.1 Machine Learning Compilers

In the past decade, artificial intelligence technologies driven by ML have made numerous breakthroughs in many fields, such as natural language processing [55], computer vision [19], and drug discovery [26]. As the key infrastructure for implementing ML algorithms, an easy-to-use and efficient ML framework is crucial. A large number of ML frameworks (and libraries) are currently on the market, including TensorFlow [5], PyTorch [45], JAX [8], and MxNet [11], providing developers with similar capabilities to train and serve models.

Meanwhile, in addition to traditional CPU and GPU, many application-specific integrated circuits for ML workloads have been developed to accelerate program execution. Typical representatives are Google TPU [25] and Hisilicon NPU [38]. Generating machine code for different ML frameworks to adapt to different types of hardware requires substantial engineering efforts, especially when the number of ML frameworks and hardware devices keeps increasing. ML compilers are proposed as the solution to this problem. Usually, the compiler frontend will transform source code written with the existing frameworks into hardware-independent IR, and the compiler backend will further transform IR into hardware-dependent machine code. With ML compilers, frontend frameworks only need to focus on generating IR, and backend hardware vendors only need to pay attention to supporting IR instructions.

The IR used in ML compilers is typically expressed as a computation graph (i.e., a directed acyclic graph). Graph nodes are ML operations (such as matrix multiplication and convolution) whose input and output are tensors (i.e., multi-dimensional arrays). Graph edges show the data dependencies between operations. One widely-used ML compiler is Google's XLA [4]. XLA defines its IR as HLO (High-Level Operations) to represent computation graphs. A series of frontends, including TensorFlow, PyTorch, and JAX, support XLA. ML programs written in these frameworks can be compiled into HLO. After performing hardware-independent and hardware-dependent optimizations, HLO is finally lowered to machine code by the XLA backend to run on the CPU, GPU, or TPU.

2.2 Secure Multiparty Computation

MPC originates from the Yao's Millionaires' problem [59] in the 1980s, where two rich people want to compare their wealth without giving away the exact value. Beyond this, MPC has shifted from an academic theory to practical usage in more complicated tasks, such as training ML models [30, 44, 56].

One fundamental technique used in MPC is secret sharing [6, 51]. A secret value is divided into multiple random shares and distributed to several parties. Each party only gets a subset of the shares and cannot reconstruct the original value independently. These parties jointly compute pre-defined computations (e.g., ML training) without leaking any sensitive information of the inputs or intermediate computation results. Usually, the final computation result (e.g., the trained model) are revealed to some designated parties. At that time, all parties put together their holding shares to reconstruct the result.

The private inputs, including integer and boolean values, are typically encoded over an algebraic ring or finite field. For integers, arithmetic secret sharing encrypts a secret over the ring $\mathbb{Z}_{2^k}$ and supports efficient arithmetic operations, including additions and multiplications. Correspondingly for boolean values, binary secret sharing provides a scheme to encrypt a secret over the ring $\mathbb{Z}_2$ and supports more efficient boolean operations, including XOR and AND computations. Addition (resp., XOR) operation of two arithmetic (resp., boolean) secret shares is equal to add (resp., XOR) the share of the two secrets locally. Operations with similar local-computation properties include a secret value adding a public value and a secret value multiplying a public value. In contrast, the multiplication (resp., AND) of two secret values is more complicated, which requires additional communication among the participating parties to exchange extra information. The heavy communication overhead has weakened the performance of MPC, especially in handling complex computations in real-world scenarios.

To improve the efficiency, mixed-protocols [15, 43, 46] that use arithmetic and binary secret sharing interchangeably shed light on MPC. With dedicated protocols, arithmetic and binary secret shares can be transformed back and forth to handle complex computations, including both arithmetic and non-arithmetic computations. However, these conversions also need communication. Despite the great efforts that have been made, the performance of MPC operations is still heavily communication-bound and sensitive to the network environment. Such characteristic makes MPC significantly different from traditional plaintext computations over CPU.

Besides integer and boolean operations, MPC also supports decimal computations, which are common in ML. It is more common and efficient to encode decimals as fixed-point numbers, which can be interpreted as the integer value multiplying a scaling factor. This factor is configurable and indicates how many bits represent the fractional part, and the remaining bits (except the sign bit) represent the integer part. When a fixed-pointed value multiplies another fixed-pointed value, the fractional bits double. In order to maintain that the result has consistent fractional bits with the input, a truncation operation is required.

The addition and multiplication of integers, fixed-point numbers, and boolean logic operations constitute MPC's most fundamental building blocks. ML scenarios require more complex and high-level operations. For linear operations such as matrix multiplication and convolution, a combination of those basic building blocks can accomplish them. For non-linear operations such as activation functions, we can approximate these functions using mathematical algorithms like Newton-

Raphson method [60]. Based on the linear and non-linear operations, complex ML tasks can be completed.

3 System Design

We describe the detailed system design of SPU in this section. The threat model of SPU is given first, followed by an overview of SPU's architecture and individual descriptions of SPU's main components, including programming interfaces, PPHLO, frontend, and backend.

3.1 Threat Model

SPU follows a standard MPC threat model and runs predefined programs among multiple parties, protecting input data and all intermediate results, typically only revealing the final results to some designated parties. Taking ML model training as an example, MPC can protect participating parties' training datasets and intermediate results like gradients, and reveal the trained model weights as the final results.

Furthermore, as an MPC computing engine, SPU is not restricted to any specific MPC threat models, such as the number of participating parties or if participants behave honestly [39]. The underlying MPC protocol used in SPU is configurable, allowing the threat model of the entire SPU system to be determined by the selected MPC protocol at runtime. For instance, using the semi-honest (with honest majority) ABY3 [43] protocol in SPU indicates SPU inherits ABY3's threat model, i.e., all participants follow the protocol honestly but may attempt to gain additional information from exchanged messages.

3.2 Architecture Overview

The goal of SPU is to run ML programs in a privacy-preserving manner. To complete this goal, we propose SPU's architecture, as illustrated in Figure 1. In the rest of this section, we use JAX as an example ML framework to describe SPU's design although SPU is not limited to JAX. We give evaluations on SPU's support to other frameworks in Section 4.2.3. Given an ML program written in JAX, our programming interfaces will implicitly call JAX API to convert this program into HLO (Section 2.1). This HLO graph and data visibility defined by users will be passed to SPU frontend, which compiles HLO to SPU's customized IR, i.e., PPHLO. After generating PPHLO, the frontend will further perform MPC-specific optimizations. The optimized PPHLO will then be sent to SPU backend, a virtual device built on multiple networked computing nodes. These nodes host SPU runtime responsible for executing MPC operations, and their number should match the supported parties of the configured MPC protocol.

SPU employs the SPMD (Single-Program-Multiple-Data) programming model. All nodes receive the same PPHLO to execute. The data consumed by each node are secret shares

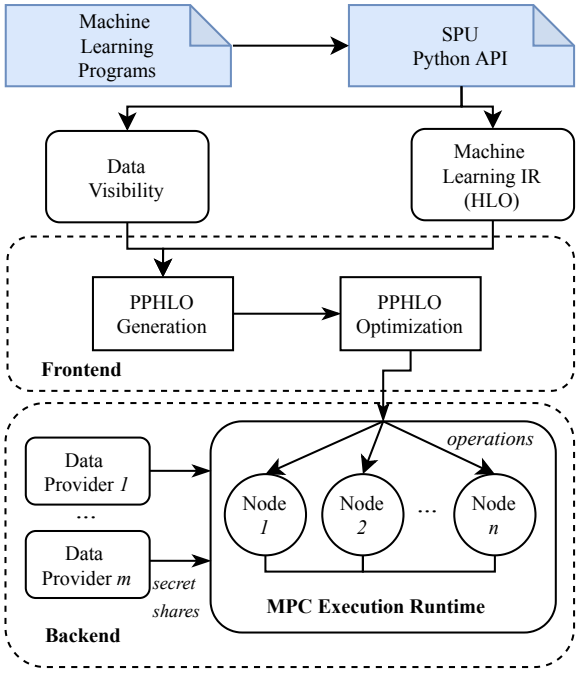


Figure 1: SPU architecture.

derived from original data held by data providers. Following the workflow described above, sensitive data from different sources can be jointly used to finish a PPML job by simply running a JAX program on SPU.

The design of SPU has the following advantages:

- By supporting ML programs of mainstream ML frameworks, SPU is extremely easy to use. SPU does not require users to learn a new library or language, or have MPC expertise.
- SPU frontend consumes HLO generated by ML programs rather than the source code. This design choice makes SPU can support a series of existing frameworks that have a lowering path to HLO directly. Moreover, SPU can benefits from platform-independent optimizations from existing ML compilers.
- SPU backend is also extensible. We can implement multiple pluggable MPC protocols in the backend without modifying PPHLO and frontend programs.
- PPHLO allows SPU to do systematic optimizations at the granularity of the high-level computational graph, which enables it to generate high-performance code for MPC execution.

3.3 Programming Interface

In order to achieve ease of use, we provide simple Python APIs so that developers can run ML programs on SPU with

```
1  import jax.numpy as jnp
2  import numpy as np
3  import spu.binding.util.distributed as ppd
4
5  # init SPU backend nodes
6  with open("/path/to/config", 'r') as file:
7      conf = json.load(file)
8  ppd.init(conf["nodes"], conf["devices"])
9
10 # specify data visibility
11 @ppd.device("P1")
12 def data_from_alice():
13     return np.random.randint(100, size=(4,))
14
15 # specify data visibility
16 @ppd.device("P2")
17 def data_from_bob():
18     return np.random.randint(100, size=(4,))
19
20 # specify a private function
21 @ppd.device("SPU")
22 def compare(x, y):
23     return jnp.maximum(x, y)
24
25 # x & y will be automatically
26 # fetched by SPU (as secret shares)
27 x = data_from_alice()
28 y = data_from_bob()
29
30 # compare will be evaluated privately by SPU
31 z = compare(x, y)
32
33 # reveal the real value of z
34 print(f"z = {ppd.get(z)}")
```

Figure 2: A demonstration of how to use SPU's API run JAX programs privately. Developers use decorators to specify data visibility and functions to be protected.

a few lines of code modifications. An example is given in Figure 2. We use SPU to solve Yao's Millionaires' problem as a simple demonstration.

We assume the two participants are called Alice and Bob. In line 3, at the start of this code, we import SPU's APIs as module *ppd*. In lines 6 to 8, we initialize the backend SPU nodes and data providers (i.e., P1 and P2 are to represent Alice and Bob). The decorators in lines 11 and 16 are data visibility marks that specify these data come from P1 and P2, which means that the two functions can only be evaluated locally on P1 and P2. The derived results are private data to be protected on SPU. The decorator on line 21 is a private function mark that specifies the function *compare* is private and should be evaluated on SPU. Lines 27 to 31 compare Alice and Bob's data. Variables x and y will be automatically fetched by SPU as secret shares, and the compared result z is also secret shares. To get the plaintext result of z, developers should use *ppd.get()* to reconstruct z as shown in line 34.

As we can see, the most crucial part of SPU's APIs is the decorator *@ppd.device()*, which is used to specify protected data and private functions. In the example demonstrated in Figure 2, the private function is a JAX *maximum* function. In fact, this can be extended to more complex JAX functions, such as an ML model training function from JAX libraries. Decorator *@ppd.device()* is the entry point for using SPU as the workflow described in Section 3.2, which will trigger HLO generation, compilation to PPHLO, and PPHLO execution. We can have SPU do all the stuff in the background by putting the decorator on top of a JAX function.

3.4 Privacy-Preserving High-Level Operations

We design PPHLO based on HLO as a customized IR for SPU because HLO lacks MPC-related semantics for optimization and efficient execution. In general, PPHLO represents a computational graph consisting of a series of operations. Each operation's input and output are tensors. The tensor type system is the most significant difference between PPHLO and other ML counterparts. A tensor's type in PPHLO can be represented by a triple <*Shape, Data Type, Visibility*>. Shape is a tensor's dimensionality. As for data type, PPHLO currently supports boolean, integer, and fixed-point numbers. Visibility is a unique tensor attribute in PPHLO. It can be either secret or public. Secret means that the tensor needs to be protected, and its real value is not visible to any node in SPU backend nodes. In contrast, public means that the tensor does not need to be protected, and any backend node can get its value.

Application developers specify the visibility of PPHLO's initial input tensors. As we described in Section 3.3, the variables generated by functions with decorator *@ppd.device()* are secret tensors, such as x and y in Figure 2. Otherwise, variables are public tensors. For each operation in PPHLO, we use the following rules to determine the output's type according to the input's type (shape is not considered here as it is determined by operation semantics). 1) **Data Type Promotion:** if one of the operands is a fixed-point number, the result is also a fixed-point number; 2) **Visibility Narrowing:** if one of the operands is a secret, the result is also a secret. Based on the two rules, we can deduce the types of all tensors in PPHLO.

Figure 3 gives an example of PPHLO in static single assignment form [50]. This code snippet corresponds to the JAX *maximum* function in Figure 2. The symbol *@main* is the program entry point. Lines 1 and 2 represent the program has two input arguments and one return value. The symbol *tensor<4x!pphlo.sec<i32>>* describes a tensor whose shape is 4, and that is a 32-bit integer secret value. Inside the braces is the program body, which contains two operations, i.e., *pphlo.greater* and *pphlo.select* (lines 3 to 6). An operation's output is assigned to the symbol on the left, which can be used as the operand of subsequent operations. When all operations

```
1  func.func @main(%arg0: tensor<4x!pphlo.sec<i32>>, %arg1: tensor<4x!pphlo.sec<i32>>)
2        -> tensor<4x!pphlo.sec<i32>> {
3    %0 = "pphlo.greater"(%arg0, %arg1) : (tensor<4x!pphlo.sec<i32>>, tensor<4x!pphlo.sec<i32>>)
4        -> tensor<4x!pphlo.sec<i1>>
5    %1 = "pphlo.select"(%0, %arg0, %arg1) : (tensor<4x!pphlo.sec<i1>>, tensor<4x!pphlo.sec<i32>>,
6        tensor<4x!pphlo.sec<i32>>) -> tensor<4x!pphlo.sec<i32>>
7    return %1 : tensor<4x!pphlo.sec<i32>>
8  }
```

Figure 3: An example of PPHLO. Generated from JAX *maximum* function shown in Figure 2.

are finished, the return value is given at line 7. PPHLO operations are extended from HLO operations [3][1]. An operation can accept public or secret tensors as its operands and has corresponding plaintext or MPC computation implementations on SPU backend runtime.

3.5 Frontend

SPU frontend is responsible for PPHLO generation and optimization. The frontend first receives an ML program's HLO and initial data visibilities as inputs and applies rules proposed in Section 3.4 to deduce the entire graph's data types and visibilities. After this step, a legal PPHLO representation is generated. The frontend will further perform code optimizations to modify PPHLO. PPHLO optimizations come from this insight: *an ML computational graph generated for non-MPC hardware may not be optimal in the MPC scenario.* We propose/implement the following compilation optimizations based on analyzing initially-generated PPHLO and our MPC expertise.

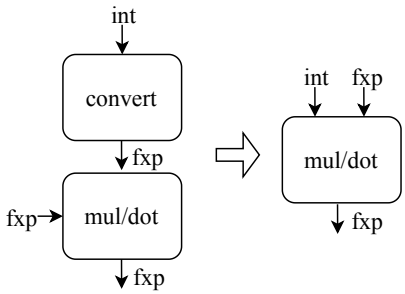


Figure 4: Mixed-data-type multiplication fusion.

Mixed-data-type multiplication fusion. In regular ML computations, when multiplying an integer to a decimal number, a *convert* operation will be called first to convert this integer to a floating-point number. Then the *multiply* operation can be dispatched to the floating-point multiplication kernel. A computation graph is illustrated in Figure 4. If we use this graph directly in SPU, an integer will be converted to a fixed-point number first, followed by a fixed-point multiplication which requires a truncation to maintain fractional bits (Section 2.2). However, an integer can directly multiply with a fixed-point number. Therefore, we can fuse the two operations into one *multiply* operation to reduce redundant truncation and conversion. This optimization applies to other similar operations like *dot*.

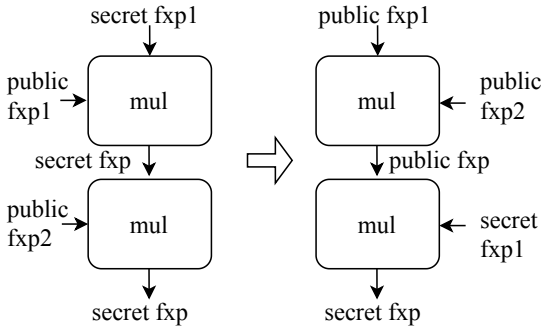


Figure 5: Mixed-visibility multiplication operands reorder.

Mixed-visibility multiplication operands reorder. Another scenario where truncation can be optimized is multiplying consecutive fixed-point numbers with mixed visibilities. As shown in Figure 5, a secret fixed-point number multiplying two public fixed-point numbers involves two *multiplication* operations. Each operation generates a secret product requiring a truncation that have a high communication overhead under some MPC protocols [24, 43]. However, we can reorder the operands without affecting the correctness. The multiplication of two public fixed-point numbers can be calculated first. The product is also public, so we can truncate the result by shifting bits locally. Then the result is used to multiply the secret fixed-point number. By reordering multiplication operands, one expensive truncation can be saved.

Inverse square root transformation. This optimization is demonstrated in Figure 6. When SPU frontend detects a computation of $y/(\sqrt{x}+u)$ where u is a tiny constant to prevent a division-by-zero problem, it will transform the computation to $y*rsqrt(x+eps())$. In the transformed computation, *eps* is

[1] Supported PPHLO operations can be found at https://github.com/secretflow/spu/blob/main/docs/reference/pphlo_op_doc.md

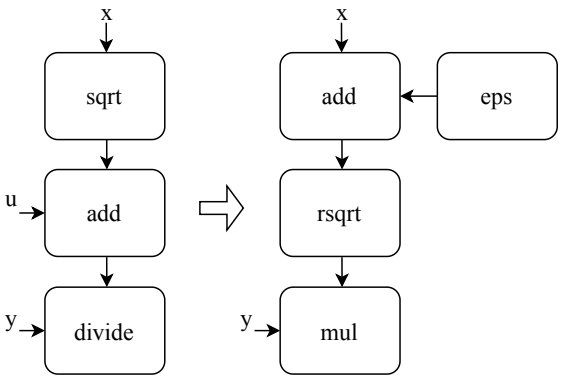


Figure 6: Inverse square root transformation.

a unique operation that will generate a minimum fixed-point number. The reason behind this transformation is that the inverse square root *rsqrt* operation has a fast MPC implementation than computing the reciprocal of the *sqrt* result (an approximation is needed). This computation pattern is observed in state-of-the-art optimizers such as Adam [31] and AMSGrad [48] when they update weights in each learning step. This optimization technique was first used by Lu et al. [41] and applied in related systems like MP-SPDZ [30] and TF Encrypted [14]. These frameworks must re-implement customized Adam and AMSGrad optimizers to employ this optimization. However, as we do the optimization at the PPHLO level, original Adam and AMSGrad optimizers in the existing JAX libraries can be directly reused.

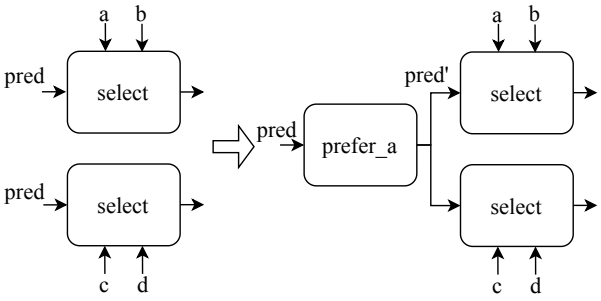


Figure 7: Select predicate reuse.

Select predicate reuse. *Select* is a commonly-used ML operation that receives a predicate and a pair of values *a* and *b*. If the predicate is true, then a is returned. Otherwise, b is returned. The predicate usually is generated with a previous comparison or logical operations and is represented as a binary secret sharing. The secret *select* operation works as a computation listed in Equation 1, where the predicate is transformed into 0 (false) and 1 (true) as a multiplier that demands converting binary secret sharing to arithmetic secret sharing. The conversion requires communication and is expensive [15]. We observe that a predicate may be used by multiple *select* operations when training some convolutional neural networks. Once SPU frontend detects this pattern, a *prefer_a* operation that explicitly converts binary secret sharing to arithmetic secret sharing will be inserted before the first select operation to reduce redundant conversions (as shown in Figure 7).

$$Select(pred, a, b) = b + pred * (a - b) \tag{1}$$

Max-pooling transformation. Max-pooling is a widely-used layer in convolutional neural networks, usually connected behind the convolutional layer for downsampling input features. In the forward propagation stage, max-pooling needs to find the maximum value in a window of values. In the backpropagation stage, max-pooling needs to replace the maximum value with its gradient while other values are set to zeros. We observe that the two stages are computed by two independent operations (i.e., *reduce_windows* and *select_and_scatter* in Figure 8) when ML frameworks train models. Although the *reduce_window* operation has found the maximum value in a window, *select_and_scatter* will do the same to find the index of the maximum value. Redundant and expensive comparisons would be called in both operations. Therefore, SPU frontend transforms this computation pattern to two new operations we proposed in PPHLO, i.e., *argmax* and *maxpool_scatter*. In the *argmax* operation, we will get the maximum value and its index in the window. The index can be directly reused by *maxpool_scatter* operation. We use a one-hot vector to represent the index. The maximum value can be updated by multiplying the index by the gradient.

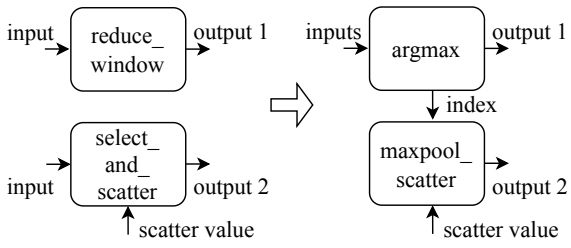


Figure 8: Max-pooling transformation.

In this section, we share observations on generated PPHLO from the ML computation graph and operation optimization strategies we implemented. Optimization techniques for ML computational graphs are much more mature with the efforts of countless experts and engineers. However, MPC computations introduce a different cost model compared to CPU or GPU computing, which enables us to adopt novel optimization techniques on PPHLO. We believe that more optimization opportunities are waiting to be discovered in this new interdisciplinary field.

3.6 Backend

SPU backend consists of multiple computing nodes, and each node contains a runtime to execute PPHLO operations. The number of nodes should match the parties of the underlying MPC protocol. The design goal of SPU backend is scalable to support different MPC protocols with a pluggable experience and efficiently execute PPHLO operations as MPC protocols. This section describes the operation dispatching mechanism and runtime optimizations used in SPU backend.

3.6.1 Operation Dispatching

We design the layered dispatching mechanism to achieve extensibility to different MPC protocols, as shown in Figure 9. For each PPHLO operation, the SPU runtime will first dispatch it to the PPHLO layer, which has a one-to-one mapping function acting as the operation entrance. The PPHLO mapping function will further dispatch the operation to fine-grained HAL (Hardware Abstraction Layer) functions. We borrow the concept of HAL from traditional operating system implementation which is mean to eliminate the boundary between hardware and software. We use HAL to hide the MPC implementation details from PPHLO operations. Specifically, at the HAL level, the SPU runtime will decompose an operation into a set of MPC-primitive functions according to the data type and visibility of operands. As the example in Figure 9, a secret fixed-point number multiplication will be decomposed to a secret integer multiplication function and a truncation function. Each MPC-primitive function will be finally dispatched to the MPC layer, corresponding to a specific implementation of fundamental MPC protocols. Adding a new MPC protocol in SPU only needs to implement the MPC-primitive function set. When users configure a new backend protocol, SPU's runtime will dispatch PPHLO operations to the new MPC implementation.

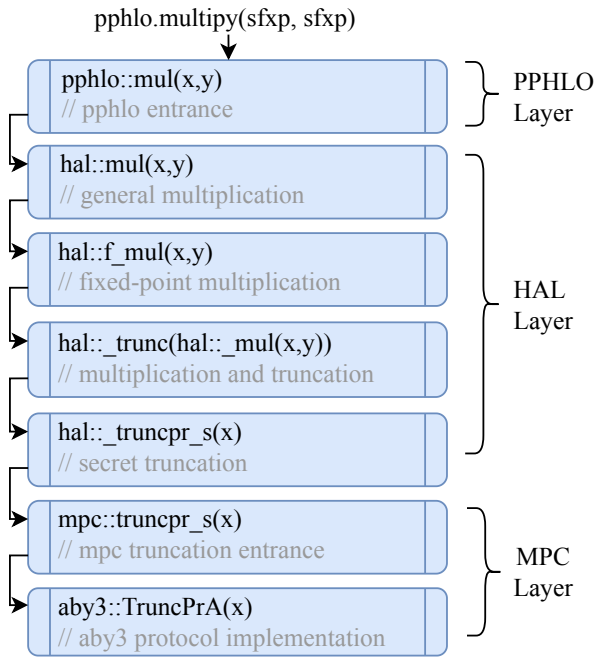


Figure 9: The dispatching path from a PPHLO operation to an MPC protocol in SPU. Different protocols can reuse the same PPHLO/HAL layer code and diverge at the final MPC layer.

3.6.2 Runtime Optimizations

We employ the following techniques in SPU runtime to achieve high performance.

Vectorization. Vectorization is a standard technique used on CPUs supported by SIMD (Single Instruction, Multiple Data) instructions. Applying one instruction to multiple data enables parallelism and improves program execution efficiency. SPU implements a similar vectorization mechanism by running one operation on a list of data to reduce the number of executed operations. For example, there are two operations *mul(a,b)* and *mul(c,d)* where *mul* stands for an element-wise multiplication operation and a, b, c, d are tensors. SPU will pack a, b, and c, d together and execute one *mul* operation on these tensors. As MPC multiplication needs communications, SPU can reduce the number of communication rounds through vectorization.

Streaming. Many MPC operations involve both intensive network I/O activities and local computations. If such an MPC operation processes a very large tensor, a more efficient method is to tile the tensor into sub-tensors and use multiple operations to process them concurrently. We illustrate this problem with the toy model in Figure 10. An ML model training stage consists of many iterations. When a processed tensor is enormous, the processing operations repeatedly block network I/O and local computing, affecting the overall execution efficiency. Suppose we tile the tensor into two small tensors and execute them concurrently. In that case, multiple sub-tensors and sub-operations can significantly improve network

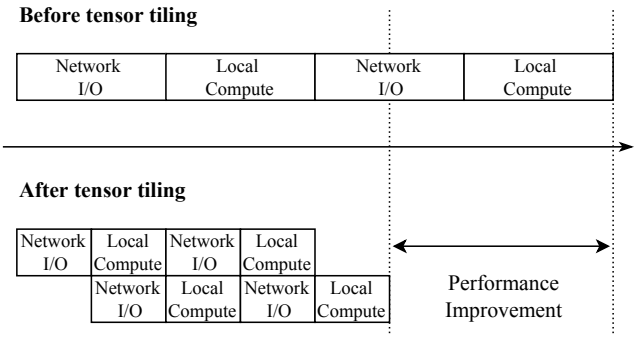


Figure 10: Streaming for MPC operations.

and computing resource utilization, thereby shortening execution time.

Concurrency. SPU supports both intra- and inter-operation concurrency to gain performance benefits. For intra-operation concurrency, most PPHLO operations consist of more than one HAL function. We implement these operations with well-established engineering experience and execute multiple functions without data dependencies in different threads. Taking the implementation of *rsqrt* operation as an example, we follow the protocol proposed by Lu et al. [41] to compute the inverse square root of a tensor x. The calculation consists of a fractional part computation and an exponential part computation which can be computed independently in two threads.

For inter-operation concurrency, SPU uses an aggressive strategy at the PPHLO graph granularity to prevent operations with communications blocking the use of computing resources. When executing PPHLO computation graph, SPU runtime launches as many operations as possible asynchronously. Once an operation's dependencies are completed, this operation will be scheduled for execution.

4 Implementation and Evaluation

We implement SPU frontend compiler based on MLIR (Multi-Level Intermediate Representation) [36], a compiler infrastructure for domain-specific computations. PPHLO is implemented as a new MLIR dialect for MPC computations. Frontend optimizations are implemented as compiler passes. We develop SPU backend with modern standard C++, and its computing nodes communicate through a high-performance RPC library, bRPC [1]. The SPU C++ library is binding to Python interfaces exposed to application developers as a Python module (Section 3.3). Currently, we provide users with three built-in MPC protocols, i.e., the semi-honest implementations of a three-party protocol ABY3 [43] and a N-party protocol SPDZ2k [13], and a two-party protocol Cheetah [24]. Our overall code base contains more than 50k LOC of C++ and 3k LOC of Python.

In the rest of this section, we evaluate SPU on performance and user-friendliness. Evaluations are done on three Alibaba Cloud *ecs.g7.xlarge* instances with 4 vCPU and 16GB RAM each. We complete evaluations under local area network (LAN, 10.1Gbps bandwidth and 0.1ms round-trip time) and wide area network (WAN, 300Mbps bandwidth and 40ms round-trip time) settings.

4.1 Performance

To evaluate SPU's performance, we compare SPU to general MPC-enabled PPML frameworks rather than some specific protocol implementations. We use SPU and three optimized frameworks (i.e., MP-SPDZ [29], TF Encrypted [14], and CrypTen [33]) to train four neural network models for image classification on the MNIST dataset. The models are trained privately with the encrypted training dataset and revealed to evaluate on the plaintext validation dataset. All frameworks train models with three ML optimizers, i.e., SGD, Adam [31], and AMSGrad [48] (except CrypTen which does not support Adam and AMSGrad). Using different optimizers to train models results in distinct computation graphs, causing varying computation costs that are noticeably divergent between MPC scenarios and plaintext training. Consequently, evaluating diverse optimizers showcases SPU's performance extensibility.

The selected four models have been widely used in related literature for evaluations [30, 40, 44, 49, 56, 57], and their detailed architectures are listed in Appendix A.1 (Table 2, 3, 4, and 5). We follow the numbering (from A to D) given by Wagh et al. [56] to refer to these models. All training experiments use a semi-honest three-party MPC (3PC) protocol, which is widely used and supported by all frameworks.

Table 1 reports the classification accuracy and seconds per batch when training 5 epochs with a batch size of 128. MP-SPDZ data is collected by running scripts provided by its author Keller [30] (commit 0f7020d). TF Encrypted data is collected by directly running scripts provided in its code repository (commit 51de98f). CrypTen data is collected by running an adaption of its official example *mpc_autograd_cnn* (commit 909df45). For SPU, we write JAX programs to train models and run these programs on SPU to collect data.

In Table 1, CrypTen has a significant gap with the other three frameworks in terms of both training speed and classification accuracy. One possible reason for this may be that CrypTen is primarily implemented in Python. Besides, CrypTen differs from the other three works in that it does not strictly employ a standard semi-honest 3PC protocol based on replicated secret sharing [6]. The protocol CrypTen uses can support any number of participants ($N \geq 2$), and we evaluate its performance in the three-party scenario.

Therefore, in order to ensure fairness, our comparisons in this section will mainly focus on SPU, MP-SPDZ, and TF Encrypted. For accuracy, the three frameworks all get high and close results. These is no single framework can achieve optimal results in every configurations. For training time, MP-SPDZ has better results under LAN while losing its advantages under WAN compared to TF Encrypted . The possible reason is that MP-SPDZ implements a more efficient multi-threaded kernel for operation execution, so it benefits from the intensive local computations under LAN. However, when network I/O communications become the bottleneck under WAN, TF Encrypted which relies on the underlying TensorFlow for graph scheduling works better.

Compared with MP-SPDZ and TF Encrypted, SPU achieves the fastest training on 11 out of 12 configurations under LAN and all configurations under WAN. Under LAN, SPU's advantage over MP-SPDZ is minor but achieves 1.4-4.6× faster training than TF Encrypted. Under WAN, SPU achieves up to 4.1× faster training than MP-SPDZ and up to

Table 1: The accuracy and seconds per batch of training four neural network models on the MNIST dataset with SGD/Adam/AMSGrad optimizer in four MPC-enabled PPML frameworks. M, T, C, and S are abbreviations of MP-SPDZ [29], TF Encrypted [14], CrypTen [33], and our SPU, respectively. CrypTen does not support Adam and AMSGrad as of the time we write this paper.

Network	Accuracy				Seconds per Batch (LAN)				Seconds per Batch (WAN)			
	M	T	C	S	M	T	C	S	M	T	C	S
A (SGD)	96.8%	96.4%	92.7%	**96.9%**	0.16	0.19	1.43	**0.12**	8.94	**4.60**	58.68	**4.60**
A (Adam)	**97.5%**	97.2%	N/A	97.4%	0.42	0.56	N/A	**0.39**	17.72	12.60	N/A	**7.67**
A (AMSGrad)	**97.6%**	97.4%	N/A	97.5%	0.42	0.71	N/A	**0.41**	18.28	13.26	N/A	**7.68**
B (SGD)	98.1%	98.3%	96.5%	**98.4%**	**1.00**	4.82	25.62	1.04	34.70	15.66	230.15	**9.87**
B (Adam)	97.9%	**98.7%**	N/A	**98.7%**	1.13	4.90	N/A	**1.12**	44.92	18.18	N/A	**11.15**
B (AMSGrad)	98.7%	**98.8%**	N/A	98.6%	1.13	4.78	N/A	**1.12**	45.73	18.08	N/A	**11.23**
C (SGD)	98.5%	**98.9%**	97.3%	98.8%	2.10	7.23	34.06	**1.81**	50.05	22.41	272.11	**12.98**
C (Adam)	98.8%	**99.0%**	N/A	98.9%	2.92	8.33	N/A	**2.37**	67.03	49.51	N/A	**22.87**
C (AMSGrad)	**99.2%**	98.9%	N/A	99.1%	2.94	8.93	N/A	**2.37**	67.49	51.06	N/A	**22.53**
D (SGD)	97.0%	**97.6%**	95.7%	97.2%	0.23	0.39	1.77	**0.22**	11.20	5.35	59.44	**4.89**
D (Adam)	97.8%	**98.0%**	N/A	97.7%	0.45	0.69	N/A	**0.43**	19.87	12.12	N/A	**7.66**
D (AMSGrad)	**98.3%**	97.5%	N/A	97.9%	0.45	0.81	N/A	**0.43**	20.42	12.76	N/A	**7.66**

2.3× faster training than TF Encrypted. Overall, the evaluation results demonstrate that SPU achieves state-of-the-art performance by combining the two aspects of WAN and LAN.

We further analyze the performance benefits of SPU. Although we implement a series of compiler passes to optimize the computation graph, it should be noted that these optimizations are workload-dependent, and not all optimizations will be effective for a specific workload. Taking training Network C with the Adam optimizer, which has the most complex computation graph, as an example. We find that when all compiler passes are disabled, the training time is 2.0× slower under LAN (4.63 versus 2.37 seconds) and is 1.9× slower under WAN (43.49 versus 22.87 seconds). Additionally, we find that nearly all performance benefits come from two frontend optimizations, i.e., max-pooling transformation and inverse square root transformation. This phenomenon does not mean that other implemented optimizations are meaningless, as we observe that those compiler passes are more effective on other workloads with corresponding computational patterns, such as training decision tree models.

Another conclusion we can draw is that the backend runtime also plays a significant role in contributing to SPU's performance improvement. Network A does not have a max-pooling layer and the SGD optimizer also does not involve the *rsqrt* operation. As a result, the two optimizations mentioned above do not apply to training Network A with SGD. However, SPU also achieves state-of-the-art in this experimental setting. Therefore, we believe SPU's high-performance benefits from collaborative frontend/backend implementations.

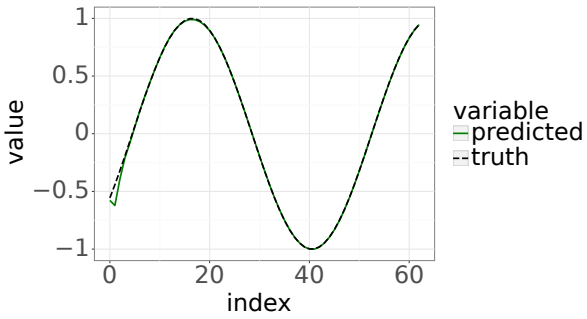
4.2 User-friendliness

This section evaluates SPU's user-friendliness through its compatibility with ML applications from different mainstream ML frameworks. We select two ML training programs from well-known open-source JAX projects and run them on SPU to train models privately. We found that only minor modifications to these programs are required to run them on SPU with acceptable overhead and achieve results comparable to plaintext training on CPUs. Besides, we test SPU's feasibility to run TensorFlow and PyTorch programs. These experiments show that SPU can be easily extended to other ML models and frameworks. The evaluations for SPU also use the same setting (a semi-honest 3PC protocol) as in Section 4.1 under LAN. The evaluations for plaintext training and prediction on CPUs run on a single cloud server.

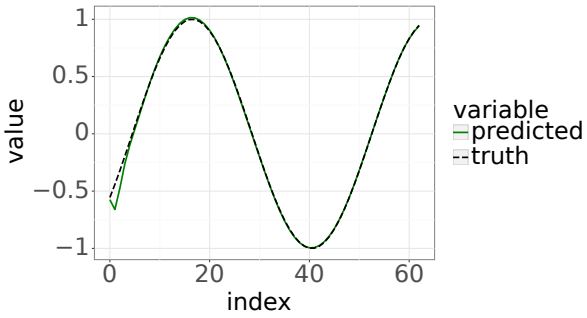
4.2.1 Long Short-Term Memory

This example of training a Long Short-Term Memory (LSTM) model [23] comes from Haiku [21], a JAX neural network library developed by DeepMind. LSTM is a recurrent neural network model for processing sequential data such as text or speech. This example trains an LSTM model to predict time series, using the data generated from a sine wave for training and validation. The model is trained privately with the encrypted training dataset and revealed to evaluate on the plaintext validation dataset. We modify about 8 lines of the example's source code to enable SPU to run the training program.

The vanilla JAX program takes 3.01 seconds to train 2001



(a) JAX.



(b) SPU.

Figure 11: Predictions with LSTM models trained by JAX and SPU.

steps, reducing the training loss from 0.45737 to 0.00067. SPU takes 7177.83 seconds (2384.7× slowdown) to reduce the training loss to 0.00099. Figure 11 shows that we use JAX and SPU-trained models to predict validation data with ground truth. The SPU-trained model achieves similar prediction results with high accuracy compared to the JAX model that is trained in plaintext.

4.2.2 Variational Auto-Encoder

This section describes a Variational Auto-Encoder [32] (VAE) model training example from Flax [20], a JAX neural network library developed by Google. VAE is a generative model which can map high-dimensional input space into low-dimensional latent space and regenerate the input from the latent representation. This example trains a VAE model to compress and regenerate images from the MNIST dataset. The model is trained privately with the encrypted training images and revealed to evaluate on the plaintext testing images. We modify about 22 lines of code to enable SPU to run the training program.

It takes JAX and SPU 214 and 9131 seconds (42.7× slowdown) to train 5 epochs with a batch size of 128,[2] reducing the training loss from 535 to 106. Figure 12 shows that using the model trained by SPU to reconstruct MNIST digits has a comparable result to JAX.

[2]The times include an extra evaluation on testing dataset in each iteration.

(a) JAX.

(b) SPU.

Figure 12: Reconstruct MNIST digits with VAE models trained by JAX and SPU. The digits above are original inputs.

4.2.3 Beyond JAX

Technically, SPU is able to support any ML frameworks that can be translated to HLO. In this section, we validate SPU's feasibility to support TensorFlow and PyTorch as frontend ML frameworks. For TensorFlow programs, SPU Python APIs can also be used directly on ML training or prediction functions made of TensorFlow functions. SPU will call *tf.function* API provided by TensorFlow to compile these composite functions into HLO as SPU frontend inputs. For PyTorch programs, SPU relies on the Torch-MLIR [2] project to convert them into HLO, which SPU can further consume.

We train a TensorFlow logistic regression model with the diagnostic Wisconsin breast cancer dataset [52]. The model is trained privately on SPU with the encrypted training dataset and revealed to evaluate on the plaintext testing dataset. SPU achieves the same ROC-AUC (Area Under the Receiver Operating Characteristic Curve) [16] of 0.99 as the plaintext training result on CPU. Training times on CPU and SPU are 0.067 and 1.121 seconds (16.7× slowdown).

As for PyTorch, we use the same dataset to train a linear classification model on the plaintext data and run predictions with jointly encrypted features on SPU (the model weights are not protected in this example). Experimental results show that compared to plaintext prediction on vanilla PyTorch, SPU achieves the same ROC-AUC of 0.97 with a 345.7× speed slowdown (0.05186 versus 0.00015 seconds). Overall, these results demonstrate SPU is feasible to support different ML frameworks.

5 Limitations and Discussion

This section discusses some known issues of SPU. SPU uses the fixed-point representation to encode decimal numbers like other MPC-based ML systems, which leads to two limitations. First, fixed-point numbers have limited precision and range compared to floating-point numbers. This problem will cause running some ML programs on SPU to get incorrect results. We can mitigate this problem by using more fractional bits to

encode fixed-point numbers in SPU. Second, some function implementations rely on floating-point number representations (such as JAX *random.normal*), which will cause SPU to compute these functions with unexpected results. We will try to provide a library overwriting these functions in the future.

Besides, SPU currently does not support secret conditions as some operations (such as *while*) may use. In most cases, these tensors are not data that need to be protected. Developers can implement these values as public tensors.

6 Related Work

In recent years, there has been a line of works studying applying MPC techniques for PPML [9,24,27,34,43,44,49,56,57]. Most of these works focus on protocol optimizations and innovations, improving MPC-enabled PPML's performance and making it a practical solution. These works diverge on security models (malicious or semi-honest adversaries, honest-majority or dishonest-majority), secret sharing schemas, the number of supported parties, and implementation details of basic operations. These protocol innovations are orthogonal to SPU, which can implement them as the underlying protocols.

Another bunch of works try to reduce MPC usage difficulty for non-MPC experts by implementing general-purpose MPC compilers. These compilers convert functions written in high-level or domain-specific languages to MPC circuits, which are later executed by backend runtime in an MPC manner. Hastings et al. [18] have a detailed survey on these compilers. However, these works are not tailored-made for ML scenarios. Using them to develop complex and efficient ML programs takes significant work.

The most relevant works to SPU are TF Encrypted [14], CrypTen [33], and MP-SPDZ [29]. TF Encrypted and CrypTen provide programming interfaces similar to TensorFlow and PyTorch in their Python modules. Refactoring an ML program into the PPML version must replace original ML APIs with TF Encrypted/CrypTen APIs corresponding to MPC implementations. The frontend of MP-SPDZ is Python. Users write ML programs based on Python APIs provided by MP-SPDZ, which compiles programs to byte code and runs in an MPC manner. Compared with these frameworks, SPU runs programs written in existing ML frameworks. Besides, SPU can support more than one ML framework.

There are other PPML frameworks developed based on Trusted Execution Environments (TEE) [47] or federated learning (FL) [28]. TEE-based solutions require special hardware and are vulnerable to side-channel attacks [10,54]. FL also enables multiple participants to jointly and privately train a model. In the classic FL scenario, each participant performs local gradient computations on the plaintext datasets, and then a centralized server aggregates the model parameters from participants. As the original input data remains within the owner's domain throughout the training process, FL can be considered as a PPML solution. However, some works have already shown that even only exchanging model parameters may also threaten the original input data [35,42,61].

Compared with FL frameworks, SPU provides end-to-end privacy protection based on provable MPC techniques. MPC does not necessarily require an independent server responsible for model aggregation. Participants' data is first encrypted and then fed into SPU. SPU then performs computations on the encrypted data (such as gradient updates) and trains a model, which is also kept in encryption. Finally, the model is reconstructed to plaintext and revealed to some designated parties. In addition to model training, another use case for SPU is private model inference, in which one party protects the input data, and the other protects the model parameters.

More advanced FL frameworks have been proposed in recent years to improve FL's security. Chen et al. [12] introduce MPC techniques into model aggregation to resist generative adversarial network attacks [22]. HybridAlpha [58] uses functional encryption [7] to prevent curious aggregators and colluding participants from inferring private data. Both approaches require a model aggregator and an additional trusted third party, which are not necessarily required in SPU. Besides, Chen et al. [12] mainly target convolutional neural networks, while SPU is not limited to specific model types. Triastcyn et al. [53] propose FedGP to replace participants' original data with artificial data by training generative adversarial networks [17]. However, their approach is limited to protecting image data and lacks a theoretical security guarantee. Compared with FedGP, SPU has no restrictions on the protected data types, and its security guarantee is based on provable MPC techniques.

7 Conclusion

In this paper, we propose SPU, a compiler and runtime suite, which converts ML programs into an MPC-specific IR and executes the IR in an MPC manner. Using the Python APIs provided by SPU, users can achieve privacy-preserving ML training and prediction by writing programs in mainstream ML frameworks. We believe that using SPU can significantly lower the threshold for users to achieve privacy protection and promote the development of the entire PPML community.

Acknowledgments

We would like to thank our shepherd, Sara Bouchenak, and the anonymous reviewers for their invaluable comments. We would also like to acknowledge the insightful feedback provided by Yu Luo on the early version of our paper. We extend our appreciation to Wen-jie Lu, Zhicong Huang, and Cheng Hong for their significant contributions to SPU. Lastly, we thank all members of the SecretFlow team for their support throughout this project.

References

[1] better rpc. https://github.com/apache/incubator-brpc, 2023.

[2] The torch-mlir project. https://github.com/llvm/torch-mlir, 2023.

[3] Xla operation semantics. https://www.tensorflow.org/xla/operation_semantics, 2023.

[4] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla, 2023.

[5] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 805–817, 2016.

[7] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *Theory of Cryptography*, pages 253–273, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[9] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511, 2019.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. *IEEE Secur. Priv.*, 18(3):28–37, 2020.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[12] Zhenzhu Chen, Anmin Fu, Yinghui Zhang, Zhe Liu, Fanjian Zeng, and Robert H. Deng. Secure collaborative deep learning against gan attacks in the internet of things. *IEEE Internet of Things Journal*, 8(7):5839–5849, 2021.

[13] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spdz2k: Efficient mpc mod 2k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 769–798, Cham, 2018. Springer International Publishing.

[14] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. Private machine learning in tensorflow using secure computation. *arXiv preprint arXiv:1810.08130*, 2018.

[15] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[16] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Commun. ACM*, 63(11):139–144, oct 2020.

[18] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: general-purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[20] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2020.

[21] Tom Hennigan, Trevor Cai, Tamara Norman, and Igor Babuschkin. Haiku: Sonnet for JAX, 2020.

[22] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: Information leakage from collaborative deep learning. In *Proceedings of*

the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, page 603–618, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[24] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure Two-Party deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[26] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

[27] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association.

[28] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *Found. Trends Mach. Learn.*, 14(1-2):1–210, 2021.

[29] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[30] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *International Conference on Machine Learning*, pages 10912–10938. PMLR, 2022.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[32] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[33] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubhabrata Sengupta, Mark Ibrahim, and Laurens van der Maaten. CrypTen: Secure multi-party computation meets machine learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[34] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *2020 IEEE Sym-*

posium on Security and Privacy (SP), pages 336–353, 2020.

[35] Maximilian Lam, Gu-Yeon Wei, David Brooks, Vijay Janapa Reddi, and Michael Mitzenmacher. Gradient disaggregation: Breaking privacy in federated learning by reconstructing the user participant matrix. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 5959–5968. PMLR, 2021.

[36] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[37] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[38] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44, 2019.

[39] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, 2021.

[40] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Wen-jie Lu, Yixuan Fang, Zhicong Huang, Cheng Hong, Chaochao Chen, Hunter Qu, Yajin Zhou, and Kui Ren. Faster secure multiparty computation of adaptive gradient descent. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, PPMLP'20, page 47–49, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 691–706. IEEE, 2019.

[43] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

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

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[46] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182. USENIX Association, August 2021.

[47] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. Securetf: A secure tensorflow framework. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, pages 44–59, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[49] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 707–721, New York, NY, USA, 2018. Association for Computing Machinery.

[50] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[51] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[52] Nick Street, William Wolberg, and O Mangasarian. Nuclear feature extraction for breast tumor diagnosis. *Proc. Soc. Photo-Opt. Inst. Eng.*, 1993, 01 1999.

[53] Aleksei Triastcyn and Boi Faltings. Federated generative privacy. *IEEE Intelligent Systems*, 35(4):50–57, 2020.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 991–1008, USA, 2018. USENIX Association.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[56] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies*, 2019.

[57] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. 2021.

[58] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19, page 13–23, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.

[60] Tjalling J. Ypma. Historical development of the newton–raphson method. *SIAM Review*, 37(4):531–551, 1995.

[61] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14747–14756, 2019.

A Supplementary Materials

A.1 Evaluated Neural Network Models

This section describes models used in Section 4.1. Network A is defined in [44]. Network B is defined in [40]. Network C is defined in [37]. Network D is defined in [49].

Table 2: Architecture of Network A (SecureML [44] model).

Layer	Input	Description	Output
Fully Connected	784	784×128 matrix multiplication	128
ReLU	128	Element-wise ReLU on input	128
Fully Connected	128	128×128 matrix multiplication	128
ReLU	128	Element-wise ReLU on input	128
Fully Connected	128	128×10 matrix multiplication	10

Table 3: Architecture of Network B (MiniONN [40] model).

Layer	Input	Description	Output
Convolution	$1 \times 28 \times 28$	5×5 kernel, 1×1 stride	$16 \times 24 \times 24$
MaxPooling	$16 \times 24 \times 24$	2×2 kernel	$16 \times 12 \times 12$
ReLU	$16 \times 12 \times 12$	Element-wise ReLU on input	$16 \times 12 \times 12$
Convolution	$16 \times 12 \times 12$	5×5 kernel, 1×1 stride	$16 \times 8 \times 8$
MaxPooling	$16 \times 8 \times 8$	2×2 kernel	$16 \times 4 \times 4$
ReLU	$16 \times 4 \times 4$	Element-wise ReLU on input	$16 \times 4 \times 4$
Fully Connected	256	256×100 matrix multiplication	100
ReLU	100	Element-wise ReLU on input	100
Fully Connected	100	100×10 matrix multiplication	10

Table 4: Architecture of Network C (LeNet [37] model).

Layer	Input	Description	Output
Convolution	$1 \times 28 \times 28$	5×5 kernel, 1×1 stride	$20 \times 24 \times 24$
MaxPooling	$20 \times 24 \times 24$	2×2 kernel	$20 \times 12 \times 12$
ReLU	$20 \times 12 \times 12$	Element-wise ReLU on input	$20 \times 12 \times 12$
Convolution	$20 \times 12 \times 12$	5×5 kernel, 1×1 stride	$50 \times 8 \times 8$
MaxPooling	$50 \times 8 \times 8$	2×2 kernel	$50 \times 4 \times 4$
ReLU	$50 \times 4 \times 4$	Element-wise ReLU on input	$50 \times 4 \times 4$
Fully Connected	800	800×500 matrix multiplication	500
ReLU	500	Element-wise ReLU on input	500
Fully Connected	500	500×10 matrix multiplication	10

Table 5: Architecture of Network D (Chameleon [49] model).

Layer	Input	Description	Output
Convolution	$1 \times 28 \times 28$	5×5 kernel, 2×2 stride	$5 \times 14 \times 14$
ReLU	$5 \times 14 \times 14$	Element-wise ReLU on input	$5 \times 14 \times 14$
Fully Connected	980	980×100 matrix multiplication	100
ReLU	100	Element-wise ReLU on input	100
Fully Connected	100	100×10 matrix multiplication	10

B Artifact Appendix

B.1 Abstract

SecretFlow-SPU is an open-source framework designed for privacy-preserving machine learning. This artifact contains

the source code of SecretFlow-SPU, along with documentation for reproducing the experiments reported in this paper. Additionally, we provide scripts and a Docker container image to quickly build the experimental settings.

B.2 Scope

The artifact includes experiments for secure neural network training, secure Variational Auto-Encoder (VAE) training, and secure Long Short-Term Memory (LSTM) training using JAX. We also provide two simple TensorFlow and PyTorch demos. These experiments cover all we reported results in the paper.

B.3 Contents

README.md describes the artifact and provides a road map for evaluation. For more details on the SecretFlow-SPU repo's directory layout, please refer to *REPO_LAYOUT.md* under the base directory.

B.4 Hosting

The artifact is available at https://github.com/secretflow/spu (branch *atc23_ae*).

B.5 Requirements

SecretFlow-SPU has no special hardware requirements. To reproduce our results, users should have at least three servers that are connected within a high-performance network. We have done our evaluations on three Alibaba Cloud ecs.g7.xlarge cloud servers with 4 vCPU and 16GB RAM each. The CPU model is Intel(R) Xeon(R) Platinum 8369B CPU @ 2.70GHz. We evaluated SecretFlow-SPU on Ubuntu 20.04.5 LTS with Linux kernel 5.4.0-125-generic. Technically, SecretFlow-SPU is supported to run on any Linux servers with software requirements described in *CONTRIBUTING.md*.