HawkEye: Statically and Accurately Profiling the Communication Cost of Models in Multi-party Learning

Wenqiang Ruan
Fudan University

Xin Lin
Fudan University

Ruisheng Zhou
Fudan University

Guopeng Lin
Fudan University

Shui Yu
University of Technology Sydney

Weili Han*
Fudan University

Abstract

Multi-party computation (MPC) based machine learning, referred to as multi-party learning (MPL), has become an important technology for utilizing data from multiple parties with privacy preservation. In recent years, in order to apply MPL in more practical scenarios, various MPC-friendly models have been proposedto reduce the extraordinary communication overhead of MPL. Within the optimization of MPC-friendly models, a critical element to tackle the challenge is profiling the communication cost of models. However, the current solutions mainly depend on manually establishing the profiles to identify communication bottlenecks of models, often involving burdensome human efforts in a monotonous procedure.

In this paper, we propose `HawkEye`, a static model communication cost profiling framework, which enables model designers to get the accurate communication cost of models in MPL frameworks without dynamically running the secure model training or inference processes on a specific MPL framework. Firstly, to profile the communication cost of models with complex structures, we propose a static communication cost profiling method based on a prefix structure that records the function calling chain during the static analysis. Secondly, `HawkEye` employs an automatic differentiation library to assist model designers in profiling the communication cost of models in `PyTorch`. Finally, we compare the static profiling results of `HawkEye` against the profiling results obtained through dynamically running secure model training and inference processes on five popular MPL frameworks, `CryptFlow2`, `CrypTen`, `Delphi`, `Cheetah`, and `SecretFlow-SEMI2K`. The experimental results show that `HawkEye` can accurately profile the model communication cost without dynamic profiling.

1 Introduction

As more and more countries publish privacy protection regulations, such as GDPR from the EU, multi-party computation (MPC) based machine learning, referred to as multi-party learning (MPL) [42, 43], has become an important technology for utilizing data from multiple parties with privacy preservation [40]. On the one hand, in industry, many IT giants have released MPL frameworks, such as `CrypTen` [25] released by Facebook, `TF-Encrypted` [9] released by Google, and `SecretFlow` [30] released by Ant group et al., to meet their data sharing requirements with high-security restrictions, where raw data must be stored distributively. For example, Ant group applies `SecretFlow` to deliver personalized policies for different customers of insurers. On the other hand, in academia, researchers have proposed dozens of MPL frameworks, such as `Cheetah` [20], `Delphi` [32], and `CRYPTGPU` [44], in recent years. However, although MPL has started to be used broadly, the huge communication overhead still seriously limits its practical applications.

In recent years, to reduce the huge communication overhead of MPL, many MPC-friendly models [11, 14, 26, 28, 37, 49] have been proposed. Within the optimization of MPC-friendly models, a critical element to tackle the challenge is profiling the communication cost of models. For example, through profiling the communication cost of the secure DenseNet-121 [19] inference process, Ganesan et al. [14] found that the convolution layer is the communication bottleneck. Then, they designed an optimized convolution operator to reduce its communication overhead. In the above process, profiling the model communication cost is a key step, which is also a popular methodology of efficiency optimization in the machine learning field [1, 48].

Even though model communication cost profiling is a key step in the design of MPC-friendly models, an effective model communication cost framework is still absent. Therefore, model designers have to manually establish the profiles to find communication bottlenecks of models, often involving burdensome human efforts in a monotonous procedure. Intuitively, model designers can profile model communication cost by inserting test instruments between each layer and dynamically running the secure model training or inference processes on a specific MPL framework [11, 14, 26, 49]. We

*Corresponding author: wlhan@fudan.edu.cn

refer to this method as dynamic profiling. Although dynamic profiling can accurately find the communication bottleneck of models, it requires specific designs and implementation for different models. As a result, dynamic profiling usually takes burdensome human efforts and computing resources, thus being less efficient. On the other hand, a few MPL frameworks, such as `SecretFlow-SPU` [30], support model communication cost profiling at the operation level, i.e., outputting the communication cost of each basic operation (e.g., multiplication, comparison). However, because `SecretFlow-SPU` uses Google's XLA [41] as a black box to compile the input program, it requires model designers to manually construct many models with a single layer (e.g., convolution) to test the communication cost of different model components. Note that it is the key to analyzing the communication cost of each layer when model designers try optimizing MPC-friendly model [15].

In this paper, we propose `HawkEye`, a static model communication cost profiling framework to accurately profile model communication cost without dynamic profiling. Firstly, to profile the communication cost of models with complex structures, we propose a static communication cost profiling method based on a prefix structure that records the function calling chain. We describe the details of the prefix structure in Section 3.2. With this method, the communication cost of models can be automatically profiled without manually inserting test instruments and dynamically running the secure model training or inference process. Because the program execution flow must be input-independent to ensure security in MPL frameworks, we can accurately know the computations required by a secure model training or inference process through such static analysis. Therefore, the static profiling results from `HawkEye` are consistent with those obtained by the dynamic profiling.

Secondly, `HawkEye` employs an automatic differentiation (Autograd) library, which integrates our proposed static communication cost profiling method, to assist model designers in profiling the communication cost of models in `PyTorch`. In particular, to avoid the bias brought by the extra communication overhead of the broadcast operator, which is commonly used to construct machine learning models, we design an optimized broadcast operator that is communication-efficient for various MPL frameworks. Based on our proposed static communication cost profiling method and the Autograd library, `HawkEye` can receive `PyTorch`-based model training or inference codes, then automatically profile model communication cost. As a result, with `HawkEye`, model designers can focus on optimizing the structure of MPC-friendly models without spending much time on profiling model communication costs.

Finally, we compare the static profiling results outputted by `HawkEye` with the dynamic profiling results obtained from five MPL frameworks, `CryptFlow2` [38], `CrypTen` [25], `Delphi` [33], `Cheetah` [20], and `SecretFlow-SEMI2K` [30]. The experimental results show that `HawkEye` can accurately profile the communication cost of models in various MPL frameworks without dynamic profiling. Furthermore, we conduct three case studies to show the practical applications of `HawkEye` (Section 5.5). For example, by applying `HawkEye` to profile the communication cost of models in four MPL frameworks with different security models, we find that under the same assumption on the number of colluded parties, MPC-friendly models optimized for MPL frameworks with semi-honest security models remain effective when model designers transfer the same optimization for MPL frameworks with malicious security models. The above results show that `HawkEye` can effectively help model designers optimize the structures of MPC-friendly models without dynamic profiling.

To summarize, we highlight our contributions as follows.

- We propose a static model communication cost profiling method to accurately analyze the communication cost of models in MPL frameworks without dynamic profiling.
- We design and implement `HawkEye` with an Autograd library to assist model designers in profiling the communication cost of models in `PyTorch`.

2 Preliminaries

2.1 Multi-Party Learning

MPL enables multiple parties to collaboratively perform model training or inference on their private data with privacy preservation. Since Mohassel and Zhang published `SecureML` [35], the pioneer study of MPL, it has become a significant topic in both industry and academia. Currently, There are mainly two technical routes to implement MPL: secret sharing [8] and homomorphic encryption (HE) [47]. Because secret sharing-based MPC protocols usually have higher efficiency in arithmetic operations, MPL frameworks mainly take secret sharing-based MPC protocols as their underlying protocols to implement secure model training or inference. HE is typically used to implement secure model inference because it can significantly reduce the computation and communication overhead of clients who hold data.

Fixed-point number representation and computation. Fixed-point number is a widely used data representation in MPL. A fixed-point number $\widetilde{x}$ with f-bit precision is encoded by mapping it to an integer $\overline{x}$, i.e., $\overline{x} = \widetilde{x} * 2^f$, where $\overline{x}$ is an element of ring or field. Since Multiplication would double the fractional part of the output, i.e., $\overline{z} = \overline{x} * \overline{y} = \widetilde{x} * \widetilde{y} * 2^{2f}$, we need to perform truncation on the output to restore the bit length of its fractional part as f.

We then introduce the basic operations that are necessary for a secret sharing-based MPL framework as follows:
Share: Given an input x, it generates m shares $\langle x \rangle_1, \cdots, \langle x \rangle_m$ and distributes them to corresponding parties.

Reveal: Given m shares $\langle x\rangle_1, \cdots, \langle x\rangle_m$, it reconstructs the original value x.
Multiplication: Given two shares $\langle x\rangle_i, \langle y\rangle_i$, it outputs $\langle z\rangle_i$ to P_i such that $z = x * y$.

The more complicated operations, such as truncation or exponentiation, can be implemented by composing the above basic operations [4, 5]. Besides combining basic operations, some MPL frameworks [10, 34, 45] provide special optimizations for complicated operations. For example, Mohassel et al. [34] design an efficient comparison protocol in `ABY3`. In this case, the complicated operations with the special optimizations can be viewed as basic operations of the corresponding MPL framework.

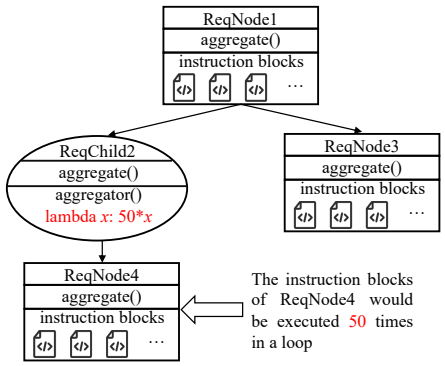


Figure 1: An example of the `MP-SPDZ` compiler's block tree. Rectangles and ovals represent ReqNodes and ReqChilds.

2.2 `MP-SPDZ` Compiler

We build upon the `MP-SPDZ` compiler to construct `HawkEye`. The `MP-SPDZ` compiler translates *mpc* programs written in Python into a sequence of instructions that represent basic operations of MPL frameworks and store instructions in instruction blocks that contain instructions without branching. Through organizing instruction blocks generated from an *mpc* program as a block tree, the `MP-SPDZ` compiler can output the number of basic operations required by the *mpc* program.

As is shown in Figure 1, the block tree of the `MP-SPDZ` compiler contains two types of nodes: ReqNode and ReqChild. ReqNode stores a list of instruction blocks. ReqChild records the control flow information. The root node of a block tree is a ReqNode. Concretely, a ReqNode instance contains an aggregate function, a list of instruction blocks, and a list of children, which could be ReqNode or ReqChild. The aggregate function of a ReqNode instance is used to count the number of basic operations in its instruction blocks and children. A ReqChild instance contains an aggregator function, an aggregate function, and a list of children, which must be ReqNode. The aggregator function of a ReqChild instance is an anonymous function that contains the control flow information. For example, in Figure 1, the aggregator function of ReqChild2 is an anonymous function that multiplies input data (i.e., the operation statistics from its children) fifty times because the instructions in ReqNode4 represent a loop body in a loop whose size is 50. The aggregate function of a ReqChild is used to sum the operation statistics outputted by its aggregator function. After the `MP-SPDZ` compiler generates the block tree from an *mpc* program, it recursively calls the aggregate functions of nodes in the block tree to compute the number of operations required by the *mpc* program.

2.3 Automatic Differentiation

Autograd automatically computes the derivative (gradients) of parameters to simplify model construction. It automatically computes the partial derivative of a function by expressing the function as a sequence of basic operators. The partial derivatives of output to the intermediate values and inputs are computed by applying the chain rule to these basic operators. Therefore, with autograd technology, model designers only need to define the forward process of models, and the gradients of model parameters can be automatically computed in the backward process. Following `PyTorch`, we design and implement the Autograd library of `HawkEye` based on operator overloading-based autograd technology.

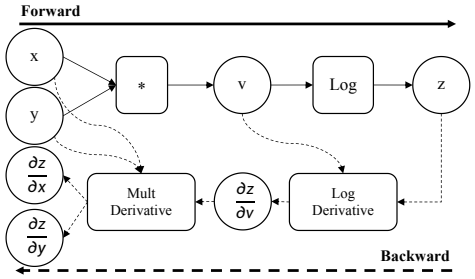


Figure 2: An example of the operator overloading-based autograd. The solid lines represent the forward process, and the dashed lines represent the backward process.

The main idea of operator overloading-based autograd technology is to overload basic operators so that each basic operator contains a forward function and a derivative computation function. When model designers use one overloaded basic operator in the forward phase, the basic operator is recorded in an operator list for derivative computations. After the forward process is finished and the backward process starts, the derivative computation function of each overloaded operator is sequentially called from the end of the operator list to compute the derivative of outputs to the intermediate values and inputs. In this way, model designers can use overloaded basic operators to construct complex functions and automatically compute the gradients of the target function. For example, as is shown in Figure 2, in the forward phase, to compute the output z, model designers first obtain v by multiplying x and y. Then, model designers obtain z by computing the logarithm

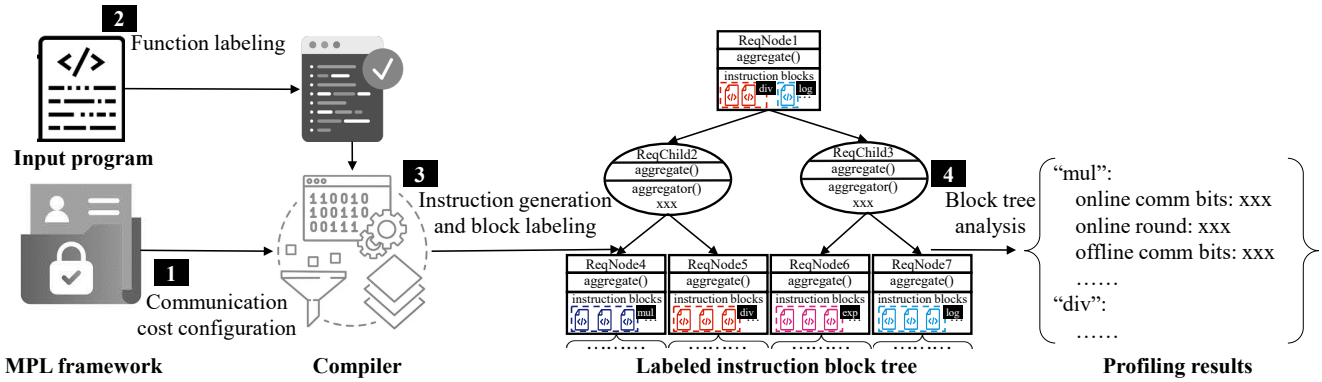


Figure 3: The workflow of our proposed static communication cost profiling method.

of v. The above two operators are recorded in a list during the forward phase. In the backward phase, the derivative of $\frac{\partial z}{\partial v}$ is computed by calling the derivative computation function of the logarithm operator with z and v as inputs. Then, $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ are computed by calling the derivative computation function of the multiplication operator with $\frac{\partial z}{\partial v}$, x and y as inputs.

3 Static Communication Cost Profiling Method

As is shown in Figure 3, the workflow of our proposed static communication cost profiling method compromises four steps: (1) Model designers choose an MPL framework to perform MPL tasks and configure the communication cost of the MPL framework with the communication cost configuration interface of `HawkEye`. (2) Given an *mpc* program, model designers label the functions in the *mpc* program they want to profile with the function labeling interface of `HawkEye`. (3) `HawkEye` generates the instruction block tree of the labeled *mpc* program and assigns each generated instruction block a label that corresponds to the labeled function based on a prefix structure. (4) `HawkEye` analyzes the labeled block tree and outputs the profiling results. Note that we have labeled forward and derivative computation functions of operators that require communication in the Autograd library of `HawkEye`. Thus, model designers can directly apply `HawkEye` to profile the model communication cost in `PyTorch` without manually labeling the model components.

3.1 Communication Cost Configuration and Function Labeling Interfaces

Communication cost configuration interface. The communication cost of one basic operation typically depends on the following five parameters: bit length k, statistical security parameter κ_s, computational security parameter κ, bit length f of a fixed-point number's fractional part, the number of parties m. The five parameters should be set according to practical requirements. As shown in Listing 1, in `HawkEye`, the communication cost of instruction is expressed as an anonymous function that takes k, κ_s, κ, f, and m as inputs and outputs a tuple that contains online communication size, online communication round, offline communication size, offline communication round. Some operations could have extra parameters. For example, the communication cost function of secure multiplication has three extra parameters: the degree of HE polynomial *deg*, HE prime coefficients modulus *mod*, and *size* that indicates the number of multiplication operations executed in parallel. We describe them in detail in the full version of this manuscript.

Listing 1. Communication cost configuration for part of basic operations of `ABY3` [34] framework. One basic operation corresponds to one basic instruction in cost_function_dict

```
class ABY3(Cost):
  cost_func_dict = {
    "share": lambda k, κs, κ, f, m: (3*k, 1, 0, 0),
    "reveal": lambda k, κs, κ, f, m: (3*k, 1, 0, 0)
    "muls": lambda k, κs, κ, f, m, size, deg, mod:
    (3*k*size, 1, 0, 0)
```

In addition to the above basic operations, complicated operations, e.g., truncation or exponentiation, are implemented by composing basic operations by default in `HawkEye`. However, if model designers use an MPL framework with special optimizations for some complicated operations (e.g., the comparison operation based on mixed protocols), they can configure the communication cost of the complicated operations. For example, many mixed-protocol MPL frameworks [10,25,33,34] implement the non-linear operations by converting arithmetic shares to boolean shares or Yao shares. To handle this type of MPL framework, model designers can specify the protocol assignments for the non-linear operations and configure their communication cost, which includes the communication cost of share conversion and boolean circuit evaluation. We further discuss how to adaptively assign protocols for non-linear operations of mixed-protocol frameworks in Section 7.

The communication cost of MPL frameworks can be configured through the following three methods: (1) Analyzing the communication costs of basic operations according to the description of the original paper. (2) Running the basic operations with open-source codes of MPL frameworks to test the communication costs of basic operations. (3) Asking the framework developers on open-source platforms to further align the communication cost configuration with the concrete MPL framework implementations. The above three methods can be solely used or combined to obtain accurate communication cost configuration. In addition, once the communication cost for one MPL framework is configured, the configuration can be shared with other users through open-source platforms.

In `HawkEye`, to assist model designers in profiling model communication cost on multiple MPL frameworks, we have configured the communication cost of ten MPL frameworks (e.g., `ABY3` [34]). Thus, model designers can directly apply `HawkEye` to profile model communication costs on the ten MPL frameworks. We show the communication cost configuration of the ten MPL frameworks in the full version of this manuscript.

Besides, when setting the value of k and f, model designers should be careful about the impact of the truncation method. In particular, local truncation, i.e., the truncation method used by `SecureML` [35] and `ABY3` [34], requires additional bits (slack bits) to maintain the desired truncation failure rate. Otherwise, the truncation results would suffer the sign bit flip error with a relatively high probability. Therefore, if using `ABY3/SecureML`-style truncation, model designers should preserve slack bits when setting the value of k and f.

Listing 2. An example program whose functions are labeled using the function labeling interface of `HawkEye`.

```
1  @buildingblock("mul") #label mul function
2  def mul(a, b):
3      return a*b
4  @buildingblock("test") #label test function
5  def test(a, b):
6      c = mul(a,b)
7      n = reveal(c)
8      return n
9  x = sint(1), y = sint(2)
10 test(x, y)
```

Function labeling interface. `HawkEye` provides a function decorator (i.e., @buildingblock) for model designers to label the functions. By adding @buildingblock("*func1*") before the declaration of a function, model designers can assign the label "*func1*" to the function. For example, as is shown in Listing 2, we assign labels "mul" and "test" to *mul* and *test* functions with the function decorator. In the main body of the program, we call the *test* function with two secret shared integers as inputs. When the bit length is 64 and model designers apply `HawkEye` to profile the program shown in Listing 2 with the communication cost configuration shown in Listing 1, the profiling result outputted by `HawkEye` would be { "initial-test": (192, 1, 0, 0), "initial-test-mul": (192, 1, 0, 0) }. The four numbers in tuples of the above dictionary represent online communication bits, online communication round numbers, offline communication bits, and offline communication round numbers, respectively. In the above dictionary, the communication cost of *test* function is the sum of all items whose labels contain "test", and the same goes for *mul* function. The format of the profiling result originated from our proposed blocking labeling methods, which are described in Section 3.2.

3.2 Block Labeling and Tree Analysis Method

After model designers label the functions in an *mpc* program with the function labeling interface of `HawkEye`, `HawkEye` generates and labels instruction blocks to obtain the labeled block tree. Then, `HawkEye` analyzes the labeled block tree to obtain the communication cost profiling results.

Block labeling method. To label instruction blocks for accurate communication cost profiling, we need to resolve the following issue: one labeled function could call another labeled function. In this situation, directly using the label of the called function to label generated instruction blocks would cause the information of the function call chain to be lost. To resolve the issue, we propose a prefix structure to record the information of the function calling relations. Specifically, we maintain a global label. When compiling a labeled function, we append the label of the function to the global label and use it to label the generated instruction blocks. After generating instructions corresponding to the function, we remove its label from the end of the global label. In this way, when a labeled function calls another labeled function, the information of the calling function would be recorded in the global label.

As shown in Algorithm 1, `HawkEye` labels instruction blocks in a recursive manner. We first define two functions, Compile_S_List and Compile_Func. Compile_S_List (Line 4-9) receives a statement list S_List, the block tree $\mathcal{T}$, and the global label g_label as inputs. It compiles the statements of S_List as instruction blocks and labels generated instruction blocks with g_label. The inputs of Compile_Func (Line 10-23) contain a function $func$, $\mathcal{T}$, g_label, and S_List. During the instruction generation process, Compile_Func enumerates statements of $func$. If one statement s does not call a labeled function f_{sub}, Compile_Func appends s into S_List (Line 19). Otherwise, Compile_Func first compiles the statements stored in the S_List (Line 14). Then, Compile_Func appends the label of f_{sub} to the end of g_label (Line 15). Without loss of generality, we assume that s is a call to f_{sub} and does not perform other computations. After that, Compile_Func calls Compile_Func with f_{sub}, $\mathcal{T}$, and g_label as inputs and then removes the label of f_{sub} from the end of g_label (Line 16-17). Compile_Func clears S_List by calling Compile_S_List (Line 22). Finally, after initializing g_label, $\mathcal{T}$, and S_List,

block labeling process is completed by calling Compile_Func with p, $\mathcal{T}$, g_label and S_List as inputs (Line 2). Note that in Algorithm 1, we view the input program p as a function that contains a sequence of statements.

Block tree analysis method. We profile the communication cost of an *mpc* program by analyzing its corresponding labeled block tree $\mathcal{T}$ outputted by Algorithm 1. As is shown in Algorithm 2, the aggregate function of a ReqNode instance receives a dictionary d_1 as the input and updates d_1 with the communication cost profiling results of the block tree whose root node is the ReqNode instance. For a ReqNode instance, its aggregate function first calculates the communication cost of each instruction block in its instruction block list (Line 5-12). Given an instruction block b, the aggregate function enumerates each instruction i in b and computes the communication cost of i by calling its communication cost function defined in the communication cost configuration dictionary (Line 8). Then, the aggregate function adds the cost function output to d_1 according to the label of b (Line 9). After calculating the communication cost of the ReqNode instance itself, its aggregate function calls the aggregate function of its children with d_1 as the input (Line 13-15).

Algorithm 1 Block labeling algorithm that generates and labels instruction blocks.

Input: An *mpc* program p that represents as a sequence of statements. The functionality *Compile_Statements* receives a statement list and the block tree $\mathcal{T}$ as input and compiles these statements into instruction blocks that are inserted into $\mathcal{T}$. The `MP-SPDZ` compiler provides it.
Output: The labeled block tree $\mathcal{T}$.

```
1:  Initialization: initialize the global label as g_label = 'initial',
    the block tree T as an empty ReqNode instance, a statement list
    S_List as [].
2:  Compile_Func(p, T, g_label, S_List)
3:  return T
4:  Function Compile_S_List(S_List, T, g_label):
5:      blocks = Compile_Statements(S_List, T)
6:      for block b in blocks do
7:          b.label = g_label
8:      end for
9:      S_List = []
10: End Function
11: Function Compile_Func(func, T, g_label, S_List):
12:     for statement s in func do
13:         if s calls a labeled function f_sub then
14:             Compile_S_List(S_List, T, g_label)
15:             g_label = g_label + '-f_sub.label'
16:             Compile_Func(f_sub, T, g_label, S_List)
17:             g_label = g_label - '-f_sub.label'
18:         else
19:             S_List.append(s)
20:         end if
21:     end for
22:     Compile_S_List(S_List, T, g_label)
23: End Function
```

The aggregate function of a ReqChild instance receives a dictionary d_1 as the input. It processes the communication cost profiling results outputted by its children with its aggregator function and updates d_1 with the process results. For a ReqChild instance, its aggregate function first initializes an empty dictionary *tmp_d* (Line 18). Then, the aggregate function of the ReqChild instance calls the aggregate functions of its children one by one with *tmp_d* as the input (Line 19-21). After that, for each pair in *tmp_d*, the aggregate function processes the item of the pair with its aggregator function and adds the result to d_1 according to the label of the pair (Line 22-24). Finally, the block tree analysis process is completed by initializing an empty dictionary d and calling the aggregate function of *root* with d as input, where *root* is the root ReqNode of the labeled block tree $\mathcal{T}$ (Line 1-3).

Algorithm 2 Block tree analysis algorithm that analyzes the labeled block tree to profile the communication cost of an *mpc* program. The addition between two tuples outputs a new tuple whose elements are the sum of two input tuples' corresponding elements

Input: The root node *root* of the labeled block tree $\mathcal{T}$ and a *cost_func_dict* stores communication cost functions of basic operations of the used MPL framework.
Output: A dictionary d that stores the profiling result.

```
1:  Initialization: Initialize an empty dictionary d
2:  root.aggregate(d)
3:  return d
4:  Function ReqNode: aggregate(self, d_1):
5:      for block b in self.blocks do
6:          for instructions i in b do
7:              if i requires communication then
8:                  i_cost = cost_func_dict[i.name](i.args)
9:                  d_1[b.label] = d_1[b.label] + i_cost
10:             end if
11:         end for
12:     end for
13:     for child c in self.children do
14:         c.aggregate(d_1)
15:     end for
16: End Function
17: Function ReqChild: aggregate(self, d_1):
18:     Initialize an empty dictionary tmp_d
19:     for node n in self.children do
20:         n.aggregate(tmp_d)
21:     end for
22:     for label, item in tmp_d do
23:         d_1[label] = d_1[label] + self.aggregator(item)
24:     end for
25: End Function
```

Accuracy analysis. Because of the security requirements of MPL, the communication cost profiling results outputted by the static communication cost profiling method are consistent with those obtained by dynamic profiling. Concretely, to rigorously guarantee the data security of MPL frameworks,

the program execution flow of MPL frameworks must be input-independent [16]. Therefore, the execution flow of each labeled function, which can be viewed as a sequence of instructions, must remain unchanged as input data changes. As a result, given an *mpc* program that contains labeled functions, as long as model designers correctly configure the communication cost for basic operations of MPL frameworks, `HawkEye` can accurately profile the communication cost of the *mpc* program.

4 Autograd Library of `HawkEye`

4.1 Architecture

As is shown in Figure 4, the Autograd library of `HawkEye` contains six modules: Secure matrix computation module, Tensor module, Functional module, NN module, Optimizers, and Dataloader. The Secure matrix computation module is the basis of the other five modules and would not be exposed to model designers. The interfaces of the other five modules are fully consistent with those of `PyTorch`. We then describe the above six modules as follows.

(1) Secure matrix computation module. This module contains secure matrix computation functions that are necessary for MPL. These functions are either our supplemented computation functions (e.g., permutation) or provided by the original `MP-SPDZ` compiler (e.g., matrix multiplication). The module is the basis of the `HawkEye` Autograd library. All of the other five modules are built on this module.

(2) Tensor module. This module contains tensor computation operators, such as element-wise operators and batch matrix-matrix products. As is mentioned in Section 2.3, each tensor computation operator of the Tensor module is overloaded to contain a forward function and a derivative computation function, such that the gradients of parameters would be automatically derived. We label the forward and derivative computation functions of each operator of this module to support model communication cost profiling.

(3) Functional module. This module contains activation functions (e.g., Relu) and functions that depend on model parameters (e.g., BatchNorm). Like the Tensor module, each operator of this module contains a labeled forward function and a labeled derivative computation function.

(4) NN module. This module contains operators (e.g., Conv2d) and containers (e.g., Sequential) for model construction. The operators of the NN module are implemented using functions from the Tensor module and the Functional module. Like `PyTorch`, model designers can also define new operators based on operators of the NN module, the Tensor module, and the Functional module.

(5) Optimizers. Optimizers contain popular optimizers in machine learning, e.g., stochastic gradient descent (SGD) and Adam [24]. Their step functions that require communication are also labeled to profile the secure model training processes.

(6) Dataloader. Dataloader shuffles the input data and converts the input data into tensors.

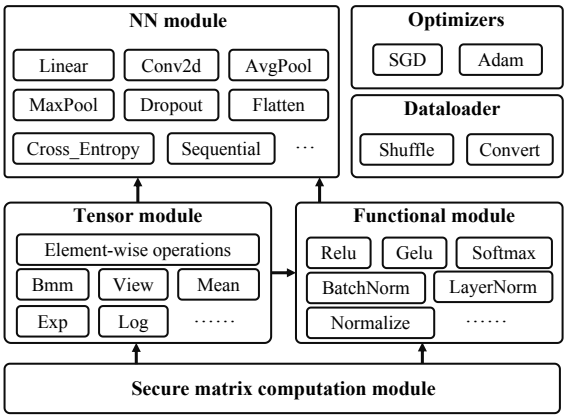
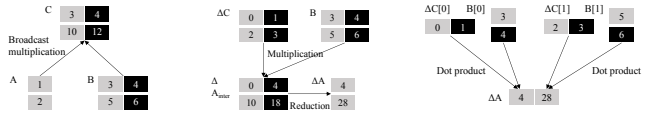


Figure 4: The architecture of the `HawkEye` Autograd library.

4.2 Optimization of Broadcast Operator

Strawman implementation of the broadcast operator. We first describe the strawman implementation of the broadcast operator and show that the implementation would bring extra communication overhead. Taking broadcast multiplication as an example, as shown in Figure 5a, given two inputs A, B with shapes (1, 2) and (2, 2), broadcast multiplication performs element-wise multiplication between each column of B and A to output C. During the backward phase, to compute the derivative of A, as is shown in Figure 5b, a straightforward method is first performing element-wise multiplication between ΔC and B to obtain an intermediate result ΔA_{inter}. Then, we can sum each column of ΔA_{inter} to reduce ΔA_{inter} as ΔA. The main drawback of the strawman implementation is that it produces the intermediate result with a larger shape than the final result, which causes extra communication overhead. The main reason is that the communication costs of some MPL frameworks, e.g., `ABY3`, only depend on the size of the computation results and are independent of the size of input data and intermediate results. Therefore, the intermediate result ΔA_{inter} whose size is two times the size of the final result ΔA would bring significantly more communication overhead, causing bias in the static profiling results from `HawkEye`.

Optimized broadcast operator. To address the drawback of the strawman implementation, we fuse the broadcast computation and intermediate result reduction as a parallel vector computation to compute the final result directly. Concretely, as shown in Figure 5c, the backward process of the optimized broadcast operator is completed by performing the dot product between each row of ΔC and B in parallel. In this way, we can avoid the intermediate result with a large size, thus



(a) Forward phase (b) Strawman backward phase (c) Optimized backward phase

Figure 5: The strawman implementation of broadcast multiplication and optimized broadcast multiplication.

avoiding extra communication overhead. For more complex broadcast cases, we can transform the input data into the two-dimensional case shown in Figure 5 through reshaping and permuting. For example, for two inputs with shapes (5, 3, 2) and (3, 2), we can reshape them as (5, 6) and (1, 6) and finish the broadcast computation in the optimized way.

4.3 Integration of Model Communication Cost Profiling Method

We then describe how the Autograd library of `HawkEye` integrates the static model communication cost profiling method introduced in Section 3. We first discuss two requirements that the Autograd library of `HawkEye` should meet to enable model designers to profile the communication cost of models in `PyTorch`. Firstly, the communication cost of the model forward and backward processes should be profiled automatically. Manually inserting test instruments in secure model training or inference codes could be time-consuming for model designers. Meanwhile, even though a few MPL frameworks (e.g., `CrypTen` [25]) provide user-friendly model construction interfaces to assist model designers in building a secure model training or inference process, they hide the derivative computation functions of operators. Thus, model designers have to modify the source codes of these MPL frameworks to profile the communication cost of model backward processes. The code modification process would require burdensome human efforts. Therefore, the Autograd library of `HawkEye` should automatically profile the communication cost of the model forward and backward processes.

Secondly, the granularity of model communication cost profiling can be adjusted without manually inserting test instruments. To simplify the model construction process, model designers usually construct models in a hierarchical manner. For example, one Densenet-121 model [19] contains multiple Bottlenecklayers and TransitionLayers. These two types of sub-modules both contain Conv2d operators, pooling operators, Relu functions, etc. To comprehensively profile the model communication cost, model designers usually need to test the communication cost of operators at different granularity.

Listing 3. The labeled exp function in the Tensor module of the Autograd library of `HawkEye`. The newly added labeling codes are highlighted on a green background and labeled with a plus sign at the beginning of the line.

```
1  + @buildingblock("exp-forward")
2  def exp(self):
3  + @buildingblock("exp-backward")
4      def propagate(dl_doutputs, operation):
5          ......
6          #The derivative computation process of exp
7          return dl_dinputs
8      ..... #The forward process of exp
9      return result
```

We meet the first requirement by labeling the forward and derivative computation functions of each operator and function that requires communication. For example, as is shown in Listing 3, the exp function in the Tensor module contains a propagate function that defines the derivative computation process of the exponentiation operator, and the rest part defines its forward process. To profile the communication cost of the forward and backward processes of the exponentiation operator, we label the exp function and its propagate function with the function labeling interface described in Section 3.1. When model designers use the exponentiation operator to construct models and profile them with `HawkEye`, the communication cost of the exponentiation operator can be obtained by summing the items whose keys contain "*exp-forward*" or "*exp-backward*" in the profiling results. Furthermore, the communication cost of the exponentiation operator forward process can be separately computed by summing the items whose labels contain "*exp-forward*". We apply the above labeling process to each operator and function that requires communication in the Autograd library of `HawkEye`. As a result, `HawkEye` can automatically profile the communication cost of the model forward and backward processes.

For the second requirement, `HawkEye` meets it based on the prefix structure of our proposed static communication cost profiling method. Concretely, model designers can adjust the granularity of model communication cost profiling by labeling the forward functions of sub-modules with the function labeling interface described in Section 3.1. Taking the Densenet-121 model as an example, as mentioned above, two sub-modules of the Densenet-121 model, i.e., Bottlenecklayers and TransitionLayers, both contain basic operators (e.g., Conv2d) of the `HawkEye` Autograd library. If model designers do not label forward functions of these two sub-modules, `HawkEye` would output the total communication cost of basic operators without distinguishing the sub-modules these basic operators belong to. In contrast, as is shown in Listing 4, if model designers label the forward function of the TransitionLayer with the label "*transitionlayer*", the communication cost of basic operators in TransitionLayer would be outputted separately with the prefix "*transitionlayer*". Because com-

posing sub-modules is a common way to construct complex models, labeling the forward functions of sub-modules should be a promising way for model designers to adjust the granularity of model communication cost profiling.

Listing 4. An example of labeling the forward functions of sub-modules to adjust the granularity of model communication cost profiling. The newly added labeling codes are highlighted on a green background and labeled with a plus sign at the beginning of the line.

```
1  class TransitionLayer(nn.Module):
2      def __init__(self, c_in, c_out):
3          super().__init__()
4          self.layers = nn.Sequential(
5              #label: transitionlayer-batchnorm
6              nn.BatchNorm2d(num_features=c_in),
7              #label: transitionlayer-relu
8              nn.ReLU(),
9              #label: transitionlayer-conv2d
10             nn.Conv2d(in_channels=c_in,
11                     out_channels=c_out, kernel_size=1),
12             #label: transitionlayer-avgpool2d
13             nn.AvgPool2d(kernel_size=2))
14 + @buildblock("transitionlayer") #labeling transitionlayer
15     def forward(self, x):
16         out = self.layers(x)
17         return out
```

5 Performance Evaluation

5.1 Implementation

We implement HawkEye by supplementing about 12k lines of code of Python (including test codes) to the MP-SPDZ compiler. At first, we modify the instruction generation process of the MP-SPDZ compiler to implement our proposed block labeling method. Then, we modify the aggregate functions of ReqNode and ReqChild in the MP-SPDZ compiler to implement our proposed block analysis method.

For the Autograd library of HawkEye, we implement all of its operators and functions according to the description of PyTorch API document [1]. Concretely, we implement functions of five PyTorch-related modules (Tensor module, Functional module, NN module, Optimizers, and Dataloader) described in Section 4.1 based on the secure fixed-point numbers computation functions provided by the MP-SPDZ compiler. The above Autograd library modules are integrated into the MP-SPDZ compiler as one of its standard libraries. Thus, in HawkEye, when model designers construct models they want to profile, they can directly import the above modules to construct models in PyTorch.

[1] https://pytorch.org/docs/stable/torch.html

5.2 Accuracy of HawkEye

Experiment Setup. We verify the accuracy of HawkEye by comparing the static profiling results outputted by HawkEye with dynamic profiling results obtained from two MPL frameworks, i.e., CypTFlow2 [38] and CrypTen [25]. We first describe two dynamic profiling processes we compare with. Firstly, Ganesan et al. [14] dynamically profile secure inference processes of four popular CNN models (i.e., DenseNet-121 [19], ResNet-50 [17], MobileNetV3 [18], ShuffleNetV2 [31]) on CrypTFlow2 [38]. Secondly, Li et al. [26] dynamically profile the secure $\text{BERT}_{\text{BASE}}$ model inference process on the two-party backend of CrypTen [25]. We run the open-source codes of the two MPL frameworks [2] [3] to obtain the dynamic profiling results. For dynamic profiling on CrypTFlow2, following Ganesan et al. [14], we set the bit length as 60 and the bit length of fixed-point numbers' fractional part as 23. For dynamic profiling on CrypTen, we set the bit length as 64 and the bit length of fixed-point numbers' fractional part as 16. We set the statistical security parameter and computation security parameter of these two MPL frameworks as 40 and 128. For the input data of CNN models, we set their sizes as $1 \times 3 \times 224 \times 224$. For the input data of $\text{BERT}_{\text{BASE}}$, following Li et al. [26], we set their sizes as $1 \times 512 \times 28996$. Because CrypTen does not support the tanh function, we replace the tanh function with the hardtanh function following Li et al. [26]. Note that because CrypTFlow2 does not have the offline phase, we only show the offline communication cost profiling results of CrypTen.

For static profiling on HawkEye, we first configure the communication cost of CryptoFlow2 [38] and CrypTen [25]. After that, we implement the models profiled by Ganesan et al. [14] and Li et al. [26] in HawkEye. Finally, we run HawkEye under the same parameter setting with CypTFlow2 and CrypTen to obtain the static profiling results.

Experimental results. As is shown in Table 1, Figure 6, Table 2, and Figure 7, HawkEye can accurately profile model communication cost on different MPL frameworks. For communication size, the proportions of operators' online/offline communication size to the total online/offline communication size outputted by HawkEye, CrypTFlow2, and CrypTen are almost the same, i.e., the proportion differences between baselines and HawkEye are all smaller than 0.50%. The slight differences between the communication size profiling results outputted by HawkEye and dynamic profiling results could be caused by the following two reasons: (1) The communication complexity of basic operations is asymptotic rather than actual. For example, CrypTFlow2 analyzes the communication complexity upper bound of truncation and comparison

[2] The code repository address of CrypTFlow2 is https://github.com/mpc-msri/EzPC. We use the codes at commit *4bae530*.

[3] The code repository address of CrypTen is https://github.com/facebookresearch/CrypTen. We use CrypTen0.4.1.

Table 1: The online communication size profiling results of four secure CNN model inference processes outputted by CryptFlow2 [38] and HawkEye. We report the proportion of each operator's online communication size to the total online communication size and the online communication size of each operator. Following Ganesan et al. [14], we list the online communication size across linear and non-linear operators.

Model	Framework	% (GB) of linear operators			% (GB) of non-linear operators
		% (GB) of Conv2d	% (GB) of other linear operators	% (GB) of all	
DenseNet-121	CryptFlow2 [38]	94.14% (646.05GB)	3.60% (24.72GB)	97.74% (670.78GB)	2.26% (15.52GB)
	HawkEye	93.92% (657.06GB) -0.22% (+11.01GB)	3.77% (26.33GB) +0.17% (+1.61GB)	97.69% (683.39GB) -0.05% (+12.61GB)	2.31% (16.17GB) +0.05% (+0.65GB)
ResNet-50	CryptFlow2 [38]	96.17% (851.76GB)	2.04% (18.05GB)	98.21% (869.81GB)	1.79% (15.83GB)
	HawkEye	96.14% (863.11GB) -0.03% (+11.35GB)	2.05% (18.43GB) +0.01% (+0.38GB)	98.19% (881.54GB) -0.02% (+11.73GB)	1.81% (16.25GB) +0.02% (+0.42GB)
MobileNet-V3	CryptFlow2 [38]	75.14% (73.30GB)	18.62% (18.16GB)	93.76% (91.46GB)	6.24% (6.09GB)
	HawkEye	74.98% (73.33GB) -0.16% (+0.03GB)	18.57% (18.16GB) -0.05% (+0.00GB)	93.55% (91.49GB) -0.21% (+0.03GB)	6.45% (6.31GB) +0.21% (+0.22GB)
ShuffleNet-V2	CryptFlow2 [38]	86.18% (38.42GB)	9.20% (4.10GB)	95.38% (42.52GB)	4.62% (2.06GB)
	HawkEye	86.14% (38.87GB) -0.04% (+0.45GB)	9.20% (4.15GB) +0.00% (+0.05GB)	95.34% (43.02GB) -0.04% (+0.50GB)	4.66% (2.10GB) +0.04% (+0.04GB)

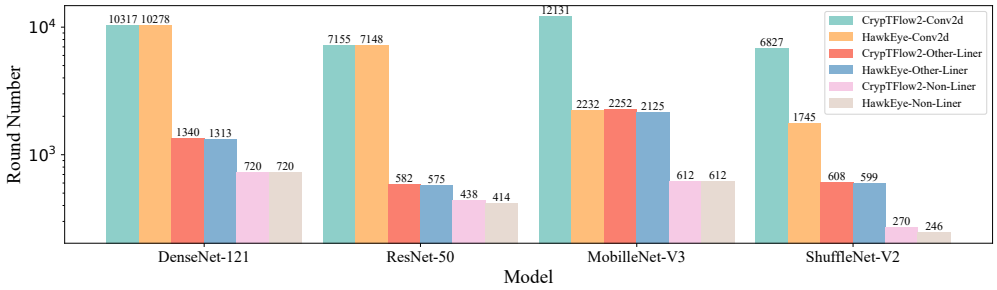


Figure 6: The online communication round profiling results of four secure CNN model inference processes outputted by CryptFlow2 [38] and HawkEye.

Table 2: The online/offline communication size profiling results of secure $\text{BERT}_{\text{BASE}}$ model inference process outputted by CrypTen [25] and HawkEye. We report the proportion of each operator's communication size to the total communication size and the communication size of each operator. Following Li et al. [26], we list the communication sizes of MatMul, Gelu, and Softmax operators.

Operator	Framework	% (GB) of online phase	% (GB) of offline phase
MatMul	CrypTen [25]	4.75% (3.22GB)	7.05% (1.37GB)
	HawkEye	4.74% (3.22GB) -0.01% (+0.00GB)	7.09% (1.37GB) +0.04% (+0.00GB)
Gelu	CrypTen [25]	26.15% (17.72GB)	23.87% (4.64GB)
	HawkEye	26.49% (18.00GB) +0.34% (+0.28GB)	24.12% (4.64GB) +0.25%(+0.00GB)
Softmax	CrypTen [25]	69.10% (46.83GB)	69.08% (13.43GB)
	HawkEye	68.77% (46.73GB) -0.33% (-0.10GB)	68.79% (13.23GB) -0.29% (-0.20GB)

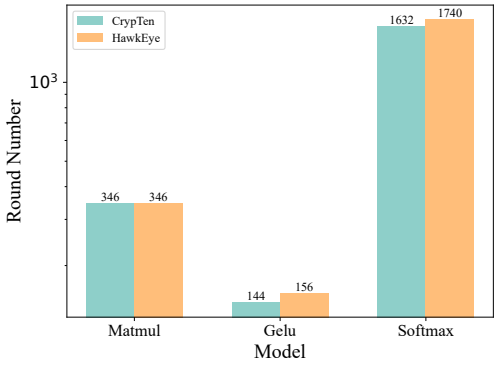


Figure 7: The online communication round profiling results of secure $\text{BERT}_{\text{BASE}}$ model inference process outputted by CrypTen [25] and HawkEye.

operations. Because we configure the communication cost for truncation and comparison operations of CryptFlow2 with the upper bound rather than the actual complexity, the communication sizes outputted by HawkEye are slightly larger than those outputted by CryptFlow2 (2) The implementations of high-level operations are different. For example, CrypTen implements the max operation by mixing the tree-based and pairwise comparison methods, while HawkEye purely uses the tree-based method. Thus, CrypTen has a larger communication size on the softmax function than HawkEye.

For communication rounds, as is shown in Figure 6 and Figure 7, HawkEye can accurately profile the communication rounds of most CNN operators on CryptFlow2 and all transformer operators on CrypTen. The main difference between the results outputted by CryptFlow2 and HawkEye lies

in the communication rounds of the Conv2d operators in MobileNet-V3 and ShuffleNet-V2 models. The main reason is that the implementation of the group convolution operator, which is used by MobileNet-V3 and ShuffleNet-V2, in CrypTFlow2 is not parallel. Therefore, the group convolution operator in CrypTFlow2 would require the group number times of communication rounds than the parallel implementation in HawkEye. The above results show that besides helping model designers analyze model communication costs, HawkEye could also help MPL framework developers find the performance issues in their implementation.

Besides CrypTFlow2 and CrypTen, we compare HawkEye with two mixed-protocol MPL frameworks (Delphi [33] and Cheetah [20]) that rely on garbled circuit (GC) and HE. Due to the space limitation, we show the experimental results in Appendix A. Furthermore, to show the accuracy of HawkEye in secure model training scenarios, we compare HawkEye with the two-party backend of SecretFlow-SPU, i.e., SecretFlow-SEMI2K, in Appendix B.

5.3 Efficiency of HawkEye

To demonstrate the efficiency of HawkEye, we compare the runtimes of HawkEye and the runtimes of CrypTFlow2 and CrypTen in the experiments of Section 5.2. All experiments are run on a Linux server equipped with two 32-core 2.30 GHz Intel Xeon CPUs and 512GB of RAM.

Table 3: Runtimes of static and dynamic profiling for five secure model inference processes in Section 5.2. We report the average results of five runs and show the standard deviations in brackets.

Network	Static profiling (s)	Dynamic profiling (s)
Densenet-121	34.38 (± 0.62)	4,429.01 (± 65.74)
Resnet-50	39.24 (± 0.13)	5,698.77 (± 80.06)
Mobilenet-V3	31.47 (± 0.31	811.92 (± 1.81)
Shufflenet-V2	15.50 (± 0.12)	319.34 (± 3.44)
$\text{BERT}_{\text{BASE}}$	471.54 (± 4.98)	618.90 (± 2.11)

As is shown in Table 3, the efficiency of HawkEye is promising for model communication cost profiling. Concretely, HawkEye can profile all CNN models within one minute. The profiling time is about eight minutes even for large $\text{BERT}_{\text{BASE}}$ model. In contrast, dynamic profiling the models using CrypTFlow2 or CrypTen requires much more time than HawkEye ($1.31\times \sim 145.23\times$). As a result, HawkEye enables model designers to efficiently profile the model communication cost and design MPC-friendly models agilely.

5.4 Ease of Use

We then report on the ease of using HawkEye to profile the communication cost of models in PyTorch by showing an example of modifying a PyTorch-based logistics regression model training codes to be HawkEye-based.

Listing 5. An example of modifying a PyTorch-based logistics regression model training codes to be HawkEye-based. The removed PyTorch codes are highlighted on a red background and labeled with a minus sign at the beginning of the line. The newly added HawkEye codes are highlighted on a green background and labeled with a plus sign at the beginning of the line.

```
1  class LogisticRegression(nn.Module):
2      def __init__(self, n_inputs, n_outputs):
3          super(LogisticRegression, self).__init__()
4          self.linear = nn.Linear(n_inputs, n_outputs)
5      def forward(self,x):
6          out = F.sigmoid(self.linear(x))
7          return out
8  - mnist = datasets.MNIST(root='./data', train=True)
9  + x = Tensor(60000, 784).get_input_from(0)
10 + y = Tensor(60000, 10).get_input_from(0)
11 - dataloader = DataLoader(mnist, batch_size=128)
12 + dataloader = DataLoader(x, y, batch_size = 128)
13 model = LogisticRegression(784, 10)
14 optimizer = optim.SGD(model.parameters(), lr = 0.01)
15 criterion = nn.CrossEntropyLoss()
16 model.train()
17 - for i in range(10):
18 + @for_range(10)
19 + def _(i):
20     x, labels = dataloader[i]
21     optimizer.zero_grad()
22     output = model(x)
23     loss = criterion(output, labels)
24     loss.backward()
25     optimizer.step()
```

As is shown in Listing 5, the model construction process, model forward process, model backward process, and model optimization process of HawkEye-based codes are fully consistent with those of PyTorch-based codes. The main differences fall on the data loading and the definition of the loop: (1) Rather than loading local data, in HawkEye, model designers need to specify the data source and then use the input data to initialize the dataloader (Line 8-12). (2) The loop interface of HawkEye slightly differs from that of PyTorch (Line 17-19). The differences should be subtle. Meanwhile, because the Autograd library of HawkEye integrates the communication cost profiling method, model designers can profile the communication cost of the secure logistics regression model training process by directly running HawkEye to analyze the program shown in Listing 5 without manually inserting test instruments. As a result, model designers can use HawkEye to profile the communication cost of complex models (e.g., transformers) in PyTorch by modifying less than ten lines of code. In contrast, Li et al. [26] have to manually insert about 100 lines of codes to dynamically profile the communication

cost of secure transformer inference processes on CrypTen. More examples of implementing complex models in HawkEye can be found in our source codes.

5.5 Case Studies

In this section, we conduct two case studies to show the practical applications of HawkEye. Firstly, to show that HawkEye can help model designers find communication bottlenecks of models on MPL frameworks with different security models, we apply HawkEye to profile model communication cost on four MPL frameworks whose security models are different. Secondly, to show that HawkEye can help model designers choose a proper optimizer for secure model training, we apply HawkEye to profile the communication cost of secure model training processes with two different optimizers.

5.5.1 Communication bottlenecks of models on MPL frameworks with different security models

Security models of MPL frameworks. We first introduce security models of MPL frameworks. The security model of an MPL framework refers to assumptions the MPL framework makes about parties. In scenarios with different security requirements, model designers usually need to employ MPL frameworks with the corresponding security models. One security model generally compromises two dimensions: the behavior of the parties and the number of colluded parties. Depending on whether parties follow the protocols, the security models of MPL frameworks can be classified as semi-honest and malicious. Depending on whether the number of colluded parties is strictly below half of the total number of parties or not, the security models of MPL frameworks can be classified as honest-majority and dishonest-majority.

We apply HawkEye to profile the secure inference processes of Resnet-50 and $\text{BERT}_{\text{BASE}}$ models on four MPL frameworks, i.e., ABY [10], SPDZ-2k [7], ABY3 [34], and Falcon [45], whose security models are (semi-honest, dishonest-majority), (malicious, dishonest-majority), (semi-honest, honest-majority), (malicious, honest-majority) respectively. For parameter settings, we set the bit length as 64, the statistical security parameter as 40, the computational security parameter as 128, the bit length of fixed-point numbers' fractional part as 16, and the number of parties as two for ABY and SPDZ-2k, three for ABY3 and Falcon. The input data sizes are consistent with those used in Section 5.2.

As is shown in Figure 8, two dimensions of the security model have significantly different impacts on communication bottlenecks of models: (1) Under the same assumption on the number of colluded parties, switching the assumption on the behavior of parties from semi-honest to malicious slightly changes the profiling results. Therefore, MPC-friendly models optimized for semi-honest MPL frameworks could remain effective for malicious MPL frameworks. (2) When underlying MPL frameworks are designed for a dishonest majority, the communication bottlenecks are linear operators, i.e., Matmul or Conv2d. In contrast, the communication bottlenecks become non-linear operators (i.e., Softmax, Relu, Gelu, MaxPool) when underlying MPL frameworks are designed for an honest majority. Therefore, model designers would need to tailor MPC-friendly models for different scenarios where the assumptions on the number of colluded parties are different. The above results show that HawkEye can help model designers efficiently find communication bottlenecks of models on MPL frameworks with different security models.

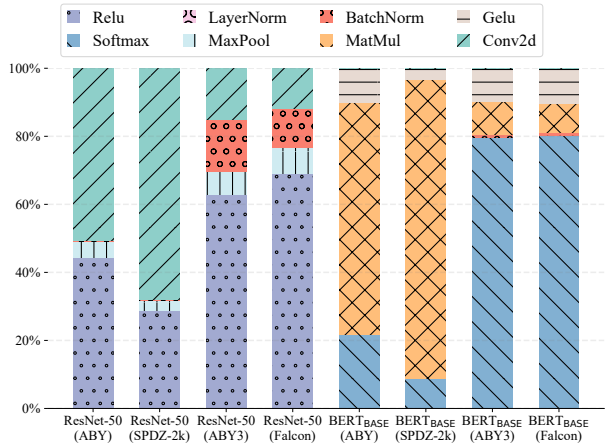


Figure 8: The proportion of each operator's online communication size to the total online communication size on four MPL frameworks with different security models.

5.5.2 Choice of optimizers in secure model training

We apply HawkEye to profile the communication cost of secure model training processes with two optimizers. A few studies [2, 29] show that in secure model training, Adam [24] could be a better optimizer than SGD because Adam can significantly improve the convergence speed. However, Adam incurs much more communication overhead than SGD. To quantitatively analyze the communication cost of optimizers, we apply HawkEye to profile the communication cost of three secure CNN model training processes (LeNet, AlexNet, VGG-16) on two MPL frameworks (ABY, ABY3). Meanwhile, following previous studies [44, 46], we replace MaxPool in the CNN models with AvgPool. Finally, we set the input data size as $128 \times 3 \times 28 \times 28$ for LeNet, $128 \times 3 \times 224 \times 224$ for AlexNet and VGG-16, where 128 is the batch size. Other parameter settings are the same as those of the first case study. Note that we run the secure model training processes on one batch of data. Because the model training process is the same for each batch of data, the proportion of each part's online communication size to the total online communication size remains unchanged as the batch number changes.

Table 4: The online communication size of gradient computation and optimization of three secure CNN model training processes. We report the proportion of each part's online communication size to the total online communication size and the online communication size of each part.

Framework	Model	Grad Computation	Optimization
ABY [10]	Lenet-SGD	100.00% (0.65GB)	0.00% (0GB)
	Lenet-Adam	92.17% (10.47GB)	7.83% (0.89GB)
	AlexNet-SGD	100.00% (12,526.23GB)	0.00% (0GB)
	AlexNet-Adam	97.06% (12,526.23GB)	2.94% (379.67GB)
	VGG-16-SGD	100.00% (189,079.51GB)	0.00% (0GB)
	VGG-16-Adam	99.55% (189,079.51GB)	0.45% (859.72GB)
ABY3 [34]	Lenet-SGD	98.94% (0.10GB)	1.06% (< 0.01GB)
	Lenet-Adam	32.26% (0.10GB)	67.74% (0.21GB)
	AlexNet-SGD	96.34% (12.12GB)	3.66% (0.46GB)
	AlexNet-Adam	3.99% (12.12GB)	96.01% (291.35GB)
	VGG-16-SGD	99.80% (521.60GB)	0.20% (1.03GB)
	VGG-16-Adam	44.15% (521.60GB)	55.85% (659.74GB)

As is shown in Table 4, the extra communication cost incurred by Adam significantly differs among different models and MPL frameworks. When training models on ABY, the online communication size of Adam only accounts for 0.45% $\sim$ 7.83% of the total online communication size. In this case, replacing SGD with Adam would significantly improve the training efficiency because the extra communication cost incurred by Adam is minor compared with the communication cost of gradient computation. In contrast, when training models on ABY3, replacing SGD with Adam would increase the total online communication size by 2.26 $\sim$ 24.12 times. Especially when securely training the AlexNet model on ABY3, replacing SGD with Adam would cause the total online communication size to increase by 24.12 times. In this case, SGD should be a better optimizer than Adam because the convergence speed improvement brought by Adam cannot cover its extra communication cost.

5.5.3 Computational graph optimization

To further show the practical application of HawkEye, we combine HawkEye with TASO [22], a classical computational graph optimization method for deep learning models, to effectively reduce the communication overhead of secure model inference. TASO improves the efficiency of secure model inference by changing the structure of the computational graphs that represent the model inference process. Meanwhile, TASO ensures that the optimized computational graph is equivalent to the original computational graph. In this experiment, we use the online communication size outputted by HawkEye as the cost model of TASO. For the underlying MPL framework, we use the three-party protocol of Deep-MPC proposed by Keller and Sun [23], whose source codes are included in MP-SPDZ, as our target MPL framework. We show the communication cost configuration of Deep-MPC in the full version of this manuscript. Meanwhile, following Jia et al. [22], we choose the backbone network (i.e., model components excluding the stem component and the final classifier) of ResNet-18 and ResNet-50 models as target models.

Table 5: The communication size and communication time of two CNN models that are optimized by original TASO and HawkEye-enhanced TASO (TASO-HawkEye). The communication time is obtained under the WAN setting where round-trip time is 72ms, and bandwidth is 9 MBps. We report the average results of five runs and show the standard deviations in brackets.

Model	Method	Comm Size (MB)	Comm Time (s)
ResNet-18	Original	286.10MB	41.30s (*pm* 0.68s)
	TASO	247.49MB	37.00s (*pm* 0.74s)
	TASO-HawkEye	210.92MB	30.54s (*pm* 1.26s)
ResNet-50	Original	1758.85MB	207.71s (*pm* 8.17s)
	TASO	1647.08MB	193.54s (*pm* 8.72s)
	TASO-HawkEye	1492.64MB	174.27s (*pm* 5.48s)

As is shown in Table 5, HawkEye-enhanced TASO significantly outperforms the original TASO. Concretely, HawkEye-enhanced TASO can reduce the communication overhead by 15.14% $\sim$ 26.28% and the communication time by 16.10% $\sim$ 26.05%, which is 1.95 $\sim$ 2.38 times and 2.36 $\sim$ 2.50 times that of the original TASO, respectively. The above results show that HawkEye can effectively help the efficiency optimization of secure model inference.

6 Related Work

Design and optimization of MPC-friendly models. Many studies [11, 14, 26, 28, 37, 39, 49] try designing and optimizing MPC-friendly models by dynamically profiling model communication cost on one or two specific MPL frameworks. Ganesan et al. [14] design an MPC-friendly convolution operator to optimize the efficiency of secure CNN model inference on CrypTFlow2 [38]. Subsequently, Peng et al. [37] propose AutoReP to automatically replace part of Relu functions of CNN models with polynomial functions to speed up secure CNN model inference on CrypTen [25]. In addition to CNN models, Liu et al. [28] design an MPC-friendly transformer model by profiling the communication cost of secure transformer inference on Cheetah [20] and CrypTFlow2 [38]. At the same time, Li et al. [26] design an MPC-friendly variant of the BERT model, namely MPCFormer, by profiling the communication cost of secure $\text{BERT}_{\text{BASE}}$ model inference process on CrypTen [25]. For the image classification task based on the transformer model, Zeng et al. [49] propose an MPC-friendly vision transformer (ViT) [12] model, namely MPCViT, by profiling the communication cost of secure ViT inference process on SecretFlow-SPU [30]. In addition, Dhyani et al. [11] design another MPC-friendly ViT model by profiling the communication cost of the secure ViT inference process on DELPHI [32]. The above studies optimize the structure of MPC-friendly models by dynamically profiling model communication cost on one or two specific

MPL frameworks. Therefore, their optimizations may only be effective in limited scenarios. With HawkEye, model designers can statically profile model communication cost on multiple MPL frameworks without implementing specific models on these MPL frameworks without dynamic profiling, thus significantly improving the automation of the optimization of MPC-friendly models.

Cost models for hybrid protocol compilers. Several studies [3, 6, 13, 21, 36] try modeling the cost of different MPC protocols to select an MPC protocol combination for efficient MPC. CheapSMC [36] models the cost of an MPC protocol by running every basic operation of the MPC protocol to obtain the cost of each basic operation. Then, they estimate the cost of an input program by counting the number of operations required by the input program. OPA [21] and HyCC [3] further improve CheapSMC by considering the circuit structure in their cost model. Concretely, in addition to testing the runtime of every single operation, they test circuits with different levels of parallelism, i.e., computing multiple operations in one communication round. Beyond its previous studies, CostCO [13] automatically generates and tests thousands of circuits with different structures to estimate the cost of MPC protocols. These studies aim to enhance the existing hybrid protocol compilers to find an efficient combination of MPC protocols for a given secure computation task. Unlike the above studies, HawkEye aims to help model designers optimize the structures of MPC-friendly models for MPL frameworks, thus improving the efficiency of MPL from the machine learning perspective. Therefore, HawkEye and existing studies on cost models for hybrid protocol compilers are orthogonal.

7 Discussion

Burdensome human efforts reduction by HawkEye. HawkEye can reduce the burdensome human efforts of dynamic profiling from two aspects: (1) model designers can statically profile the communication cost of one model on multiple MPL frameworks without manually implementing the model in the MPL frameworks. Thus, the static profiling process in HawkEye can significantly reduce human efforts. (2) HawkEye offers an Autograd library whose model construction interfaces are fully consistent with those of PyTorch and integrates our proposed static communication cost profiling method. The Autograd library of HawkEye can significantly reduce human efforts of implementing models in HawkEye and inserting test instruments.

Local computation cost optimization. Local computation cost optimization [44, 46] is another important research topic in the field of MPL. However, because communication cost is still the main performance bottleneck of MPL, we mainly consider profiling the communication cost of models in this paper. Watson et al. [46] also show that with the acceleration of GPU, even in the LAN setting, the local computation time is significantly smaller than the communication time. In the WAN setting, the local computation time is negligible compared with the communication time. Therefore, the communication cost is the main optimization goal of studies on the optimization of MPC-friendly models [14, 26, 49]. In addition, the communication optimization and local computation optimization are orthogonal. These two optimizations can improve the efficiency of MPL in different ways.

The adaptive protocol assignment for mixed-protocol frameworks. The mixed-protocol MPL frameworks (i.e., CrypTen, Cheetah, Deep-MPC and Delphi) involved in our experiments use static protocol assignments introduced in Section 3.1, i.e., model designers specify the protocol assignment and configure the communication cost of corresponding non-linear operations in HawkEye.

However, emerging mixed-protocol MPL frameworks (e.g., Silph [6]) can adaptively assign protocols to efficiently perform non-linear operations according to circuit structures. HawkEye can be extended to profile the communication cost of these frameworks that employ the adaptive protocol assignment when model designers provide protocol assignment strategies. Concretely, we can deliver the circuit structure information (e.g., the number of non-linear operators to be computed in parallel) in the parameters of the operation communication cost functions. Then, model designers can adaptively adjust the communication cost of non-linear operations according to the delivered circuit structure information.

8 Conclusion and Future Work

In this paper, we propose HawkEye, a static communication cost profiling framework to enable model designers to get the accurate communication cost of models in MPL frameworks without dynamic profiling. HawkEye contributes two folders: a static communication cost profiling method and an Autograd library. The static communication cost profiling method statically profiles model communication cost by breaking high-level components as compositions of basic operations. Meanwhile, the Autograd library has model construction interfaces that are fully consistent with those of PyTorch and integrates the static communication cost profiling method to profile the communication cost of models in PyTorch. Finally, we conduct a series of experiments to compare the static profiling results outputted by HawkEye with the dynamic profiling results from two MPL frameworks. The experimental results show that HawkEye can accurately profile the model communication cost. As a result, HawkEye can effectively help model designers optimize the structures of MPC-friendly models in MPL frameworks.

In the future, we will extend HawkEye to profile the communication costs of MPL frameworks that employ adaptive protocol assignments and the local computation cost of models in MPL frameworks, such that model designers can obtain more comprehensive information when they design and optimize MPC-friendly models.

9 Ethics considerations

We analyze the ethics of our paper from the following three principles: beneficence, respect for persons, and justice. Regarding beneficence, because we do not test any live services or APIs that give access to otherwise non-public algorithms or models, our results would not cause any financial loss. Regarding respect for persons, our study aims to improve the efficiency of MPC-friendly model development and does not have risks of disrespecting persons. Regarding justice, the results of our study would expand the practical applications of MPL and not impact any special stakeholder group.

10 Open Science

We release our codes in https://zenodo.org/records/14772876 and attend the artifact evaluation.

Acknowledgment

This paper is supported by Natural Science Foundation of China (62172100, 92370120) and the National Key R&D Program of China (2023YFC3304400). We thank Lei Kang, Xinyu Tu, Cheng Hong, Wen-jie Lu, and all anonymous reviewers for their insightful comments.

References

[1] ANDOORVEEDU, M., ZHU, Z., ZHENG, B., AND PEKHIMENKO, G. Tempo: Accelerating transformer-based model training through memory footprint reduction. In *Proceedings of Advances in Neural Information Processing Systems* (2022), S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., pp. 12267–12282.

[2] ATTRAPADUNG, N., HAMADA, K., IKARASHI, D., KIKUCHI, R., MATSUDA, T., MISHINA, I., MORITA, H., AND SCHULDT, J. C. N. Adam in private: Secure and fast training of deep neural networks with adaptive moment estimation. *Proc. Priv. Enhancing Technol. 2022*, 4 (2022), 746–767.

[3] BÜSCHER, N., DEMMLER, D., KATZENBEISSER, S., KRETZMER, D., AND SCHNEIDER, T. Hycc: Compilation of hybrid protocols for practical secure computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, p. 847–861.

[4] CATRINA, O. Round-efficient protocols for secure multiparty fixed-point arithmetic. In *Proceedings of 2018 International Conference on Communications (COMM)* (Bucharest, 2018), IEEE Press, p. 431–436.

[5] CATRINA, O., AND DE HOOGH, S. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks* (Berlin, Heidelberg, 2010), J. A. Garay and R. De Prisco, Eds., Springer Berlin Heidelberg, pp. 182–199.

[6] CHEN, E., ZHU, J., OZDEMIR, A., WAHBY, R. S., BROWN, F., AND ZHENG, W. Silph: A framework for scalable and accurate generation of hybrid mpc protocols. In *Proceedings of 2023 IEEE Symposium on Security and Privacy (SP)* (2023), pp. 848–863.

[7] CRAMER, R., DAMGÅRD, I., ESCUDERO, D., SCHOLL, P., AND XING, C. Spd$\mathbb{Z}_{2^k}$: Efficient mpc mod 2^k for dishonest majority. In *Advances in Cryptology – CRYPTO 2018* (Cham, 2018), H. Shacham and A. Boldyreva, Eds., Springer International Publishing, pp. 769–798.

[8] CRAMER, R., DAMGÅRD, I. B., AND NIELSEN, J. B. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, Cambridge, 2015.

[9] DAHL, M., MANCUSO, J., DUPIS, Y., DECOSTE, B., GIRAUD, M., LIVINGSTONE, I., PATRIQUIN, J., AND UHMA, G. Private machine learning in tensorflow using secure computation. *CoRR abs/1810.08130* (2018).

[10] DEMMLER, D., SCHNEIDER, T., AND ZOHNER, M. ABY - A framework for efficient mixed-protocol secure two-party computation. In *Proceedings of 22nd Annual Network and Distributed System Security Symposium, NDSS , February 8-11, 2015* (San Diego, California, USA, 2015), The Internet Society.

[11] DHYANI, N., MO, J., CHO, M., JOSHI, A., GARG, S., REAGEN, B., AND HEGDE, C. Privit: Vision transformers for fast private inference. *arXiv preprint arXiv:2310.04604* (2023).

[12] DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., WEISSENBORN, D., ZHAI, X., UNTERTHINER, T., DEHGHANI, M., MINDERER, M., HEIGOLD, G., GELLY, S., USZKOREIT, J., AND HOULSBY, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of International Conference on Learning Representations* (2021).

[13] FANG, V., BROWN, L., LIN, W., ZHENG, W., PANDA, A., AND POPA, R. Costco: An automatic cost modeling framework for secure multi-party computation. In *Proceedings of 2022 IEEE 7th European Symposium on Security and Privacy (Euro S&P)* (Los Alamitos, CA, USA, jun 2022), IEEE Computer Society, pp. 140–153.

[14] GANESAN, V., BHATTACHARYA, A., KUMAR, P., GUPTA, D., SHARMA, R., AND CHANDRAN, N. Efficient ml models for practical secure inference. *arXiv preprint arXiv:2209.00411* (2022).

[15] GHODSI, Z., VELDANDA, A. K., REAGEN, B., AND GARG, S. Cryptonas: Private inference on a relu budget. *Advances in Neural Information Processing Systems 33* (2020), 16961–16971.

[16] GOLDREICH, O. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1987), STOC '87, Association for Computing Machinery, p. 182–194.

[17] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778.

[18] HOWARD, A., SANDLER, M., CHU, G., CHEN, L.-C., CHEN, B., TAN, M., WANG, W., ZHU, Y., PANG, R., VASUDEVAN, V., ET AL. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision* (2019), pp. 1314–1324.

[19] HUANG, G., LIU, Z., VAN DER MAATEN, L., AND WEINBERGER, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (July 2017).

[20] HUANG, Z., JIE LU, W., HONG, C., AND DING, J. Cheetah: Lean and fast secure Two-Party deep neural network inference. In *Proceedings of 31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association.

[21] ISHAQ, M., MILANOVA, A. L., AND ZIKAS, V. Efficient mpc via program analysis: A framework for efficient optimal mixing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 1539–1556.

[22] JIA, Z., PADON, O., THOMAS, J., WARSZAWSKI, T., ZAHARIA, M., AND AIKEN, A. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 47–62.

[23] KELLER, M., AND SUN, K. Secure quantized training for deep learning. In *Proceedings of the 39th International Conference on Machine Learning* (17–23 Jul 2022), K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162 of *Proceedings of Machine Learning Research*, PMLR, pp. 10912–10938.

[24] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. In *Proceedings of 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[25] KNOTT, B., VENKATARAMAN, S., HANNUN, A. Y., SENGUPTA, S., IBRAHIM, M., AND VAN DER MAATEN, L. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual* (2021), M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, Eds., pp. 4961–4973.

[26] LI, D., WANG, H., SHAO, R., GUO, H., XING, E., AND ZHANG, H. MPCFORMER: FAST, PERFORMANT AND PRIVATE TRANSFORMER INFERENCE WITH MPC. In *Proceedings of The Eleventh International Conference on Learning Representations* (2023).

[27] LIU, J., JUUTI, M., LU, Y., AND ASOKAN, N. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 619–631.

[28] LIU, X., AND LIU, Z. Llms can understand encrypted prompt: Towards privacy-computing friendly transformers. *arXiv preprint arXiv:2305.18396* (2023).

[29] LU, W.-J., FANG, Y., HUANG, Z., HONG, C., CHEN, C., QU, H., ZHOU, Y., AND REN, K. Faster secure multiparty computation of adaptive gradient descent. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice* (New York, NY, USA, 2020), PPMLP'20, Association for Computing Machinery, p. 47–49.

[30] MA, J., ZHENG, Y., FENG, J., ZHAO, D., WU, H., FANG, W., TAN, J., YU, C., ZHANG, B., AND WANG, L. SecretFlow-SPU: A performant and User-Friendly framework for Privacy-Preserving machine learning. In *Proceedings of 2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 17–33.

[31] MA, N., ZHANG, X., ZHENG, H.-T., AND SUN, J. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)* (September 2018).

[32] MISHRA, P., LEHMKUHL, R., SRINIVASAN, A., ZHENG, W., AND POPA, R. A. Delphi: A cryptographic inference service for neural networks.

In *Proceedings of 29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 2505–2522.

[33] MISHRA, P., LEHMKUHL, R., SRINIVASAN, A., ZHENG, W., AND POPA, R. A. Delphi: A cryptographic inference system for neural networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice* (2020), pp. 27–30.

[34] MOHASSEL, P., AND RINDAL, P. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, p. 35–52.

[35] MOHASSEL, P., AND ZHANG, Y. Secureml: A system for scalable privacy-preserving machine learning. In *Proceedings of 2017 IEEE Symposium on Security and Privacy, SP 2017* (San Jose, CA, USA, 2017), IEEE Computer Society, pp. 19–38.

[36] PATTUK, E., KANTARCIOGLU, M., ULUSOY, H., AND MALIN, B. Cheapsmc: A framework to minimize secure multiparty computation cost in the cloud. In *Data and Applications Security and Privacy XXX* (Cham, 2016), S. Ranise and V. Swarup, Eds., Springer International Publishing, pp. 285–294.

[37] PENG, H., HUANG, S., ZHOU, T., LUO, Y., WANG, C., WANG, Z., ZHAO, J., XIE, X., LI, A., GENG, T., MAHMOOD, K., WEN, W., XU, X., AND DING, C. Autorep: Automatic relu replacement for fast private network inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (October 2023), pp. 5178–5188.

[38] RATHEE, D., RATHEE, M., KUMAR, N., CHANDRAN, N., GUPTA, D., RASTOGI, A., AND SHARMA, R. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2020), CCS '20, Association for Computing Machinery, p. 325–342.

[39] RUAN, W., XU, M., FANG, W., WANG, L., WANG, L., AND HAN, W. Private, Efficient, and Accurate: Protecting Models Trained by Multi-party Learning with Differential Privacy . In *2023 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, May 2023), IEEE Computer Society, pp. 1926–1943.

[40] RUAN, W., XU, M., JIA, H., WU, Z., SONG, L., AND HAN, W. Privacy compliance: Can technology come to the rescue? *IEEE Security & Privacy 19*, 4 (2021), 37–43.

[41] SABNE, A. Xla : Compiling machine learning for peak performance, 2020.

[42] SONG, L., WANG, J., WANG, Z., TU, X., LIN, G., RUAN, W., WU, H., AND HAN, W. pmpl: A robust multi-party learning framework with a privileged party. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2022), CCS '22, Association for Computing Machinery, p. 2689–2703.

[43] SONG, L., WU, H., RUAN, W., AND HAN, W. Sok: Training machine learning models over multiple sources with privacy preservation. *arXiv preprint arXiv:2012.03386* (2020).

[44] TAN, S., KNOTT, B., TIAN, Y., AND WU, D. J. CRYPTGPU: Fast privacy-preserving machine learning on the gpu. In *Proceedings of IEEE Symposium on Security and Privacy (SP)* (2021).

[45] WAGH, S., TOPLE, S., BENHAMOUDA, F., KUSHILEVITZ, E., MITTAL, P., AND RABIN, T. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proc. Priv. Enhancing Technol. 2021*, 1 (2021), 188–208.

[46] WATSON, J.-L., WAGH, S., AND POPA, R. A. Piranha: A GPU platform for secure computation. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association, pp. 827–844.

[47] YI, X., PAULET, R., AND BERTINO, E. Homomorphic encryption. In *Homomorphic Encryption and Applications*. Springer, 2014, pp. 27–46.

[48] YU, G. X., GROSSMAN, T., AND PEKHIMENKO, G. Skyline: Interactive in-editor computational performance profiling for deep neural network training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2020), UIST '20, Association for Computing Machinery, p. 126–139.

[49] ZENG, W., LI, M., XIONG, W., TONG, T., LU, W.-J., TAN, J., WANG, R., AND HUANG, R. MPCViT: Searching for Accurate and Efficient MPC-Friendly Vision Transformer with Heterogeneous Attention . In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)* (Los Alamitos, CA, USA, Oct. 2023), IEEE Computer Society, pp. 5029–5040.

Table 6: The online communication size profiling results of two secure CNN model inference processes outputted by Delphi [33] and HawkEye.

Model	Framework	% (MB) of linear operators			% (MB) of non-linear operators
		% (MB) of Conv2d	% (MB) of other linear operators	% (MB) of all	
MiniONN	Delphi [33]	0.50% (0.98MB)	0.01% (0.01MB)	0.51% (0.99MB)	99.49% (195.41MB)
	HawkEye	0.44% (0.87MB) -0.06% (-0.11MB)	0.01% (0.01MB) -0.00% (-0.00MB)	0.45% (0.88MB) -0.06% (-0.11MB)	99.55% (195.41MB) +0.06% (+0.00MB)
ResNet-32	Delphi [33]	0.75% (2.59MB)	0.00% (0.01MB)	0.75% (2.60MB)	99.25% (342.25MB)
	HawkEye	0.73% (2.52MB) -0.02% (-0.07MB)	0.00% (0.01MB) +0.00% (-0.00MB)	0.73% (2.53MB) -0.02% (-0.07MB)	99.27% (342.25MB) +0.02% (+0.00MB)

Table 7: The offline communication size profiling results of two secure CNN model inference processes outputted by Delphi [33] and HawkEye.

Model	Framework	% (MB) of linear operators			% (MB) of non-linear operators
		% (MB) of Conv2d	% (MB) of other linear operators	% (MB) of all	
MiniONN	Delphi [33]	2.34% (75.52MB)	0.03% (1.00MB)	2.37% (76.52MB)	97.63% (3150.27MB)
	HawkEye	2.00% (64.29MB) -0.34% (-11.23MB)	0.03% (0.85MB) +0.00% (-0.15MB)	2.03% (65.14MB) -0.34% (-11.38MB)	97.97% (3148.73MB) +0.34% (-1.54MB)
ResNet-32	Delphi [33]	3.70% (212.04MB)	0.02% (1.00MB)	3.72% (213.04MB)	96.28% (5517.63MB)
	HawkEye	3.43% (195.86MB) -0.27% (-16.18MB)	0.01% (0.85MB) -0.01% (-0.15MB)	3.44% (196.71MB) -0.28% (-16.33MB)	96.56% (5516.39MB) +0.28% (-1.24MB)

A Accuracy of HawkEye on Mixed-protocol Frameworks

Experiment Setup. In order to further show that HawkEye works well with mixed-protocol frameworks that rely on HE or GC, we compare the communication cost profiling results outputted by HawkEye and two MPL frameworks (i.e. Delphi [33] and Cheetah [20]) that rely on HE and GC respectively. We run two secure CNN models (MiniONN [27] and ResNet-32 [17]) inference processes on the open-source codes [4] [5] of these two MPL frameworks to obtain the dynamic profiling results. We choose MiniONN [27] and ResNet-32 [17] because these two models are tested in the paper of Delphi and Cheetah simultaneously. Because Delphi merges the batch normalization layer and the convolution layer into a single convolution operation, we omit the batch normalization layer of these two models. For parameter setting, we set the bit length as 64 and the bit length of fixed numbers' fractional part as 18. We set the statistical security parameter and computation security parameter of these two MPL frameworks as 40 and 128, respectively. Besides, we set the polynomial degree and modulus coefficients of BFV HE used by Delphi as 8192 and {43, 43, 44, 44, 44} respectively. For the CKKS HE used by Cheetah, we set the polynomial degree and modulus coefficients as 8192 and {59, 55, 49, 49} respectively. For the input data, following the setting of Delphi, we set the input data size as $1 \times 3 \times 32 \times 32$.

To obtain the static model communication cost profiling results from HawkEye, we configure the model communication cost of Delphi and Cheetah in HawkEye and implement the above models. Finally, we run HawkEye under the same parameter setting with Delphi and Cheetah to obtain the static profiling results. We also show the communication cost configuration of Delphi and Cheetah in the full version of this manuscript.

Experimental results. As is shown in Table 6, Table 7, and Table 8, HawkEye can accurately profile the model communication cost on Delphi and Cheetah. Concretely, the proportions of operators' online/offline communication size to the total online/offline communication size outputted by HawkEye and baselines are very close. The differences between the communication size profiling results outputted by HawkEye and baselines could be caused by the following two reasons: (1) The HE packing is usually not compact in the implementation. Delphi and Cheetah both use HE to compute linear operators. HE packs multiple plaintext data into one ciphertext to reduce the communication overhead. However, due to implementation issues, the HE packing usually cannot be as compact as the theoretical. Therefore, the communication sizes of linear operators outputted by Delphi and Cheetah are slightly larger than those outputted by HawkEye (2) The complexity of the comparison operations is asymptotic rather than actual. Similar to CrypTFlow2, Cheetah only provides an upper bound of the communication complexity for its comparison operators. Therefore, the communication sizes of non-linear operators outputted by HawkEye are slightly larger than those outputted by Cheetah.

B Accuracy of HawkEye on Secure Model Training Scenarios

Experiment Setup. To show that HawkEye works well in secure model training scenarios, we compare

[4] The code repository address of Delphi is https://github.com/mc2-project/delphi. We use codes at commit *92bc007*

[5] The code repository address of Cheetah is https://github.com/secretflow/spu. We use SecretFlow-SPU0.9.3b0

Table 8: The online communication size profiling results of outputted by Cheetah [20] and HawkEye for two secure CNN model inference processes.

Model	Framework	% (MB) of linear operators			% (MB) of non-linear operators
		% (MB) of Conv2d	% (MB) of other linear operators	% (MB) of all	
MiniONN	Cheetah [20]	53.99% (20.14MB)	2.41% (0.90MB)	56.40% (21.04MB)	43.60% (16.26MB)
	HawkEye	50.47% (18.34MB) -3.52% (-1.80MB)	2.26% (0.82MB) -0.15% (-0.08MB)	52.73% (19.16MB) -3.67% (-1.88MB)	47.27% (17.18MB) +3.67% (+0.92MB)
ResNet-32	Cheetah [20]	64.40% (53.03MB)	1.02% (0.84MB)	65.42% (53.87MB)	34.58% (28.47MB)
	HawkEye	63.14% (52.87MB) -1.26% (-0.16MB)	0.92% (0.77MB) -0.10% (-0.07MB)	64.06% (53.64MB) -1.36% (-0.23MB)	35.94% (30.10MB) +1.36% (+1.63MB)

Table 9: The online communication size profiling results of two secure CNN model training processes outputted by SecreFLow-SEMI2K [30] and HawkEye.

Model	Framework	% (GB) of Model Forward	% (GB) of Model Backward	% (GB) of Model Update
VGG-16	SecretFlow-SEMI2K [30]	45.59% (120.75GB)	54.28% (143.78GB)	0.13% (0.35GB)
	HawkEye	44.02% (120.04GB) -1.57% (-0.71GB)	55.85% (152.29GB) +1.57% (+8.51GB)	0.13% (0.35GB) +0.00% (+0.00GB)
ResNet-50	SecretFlow-SEMI2K [30]	47.49% (16.92GB)	51.89% (18.49GB)	0.62% (0.22GB)
	HawkEye	45.69% (16.77GB) -1.80% (-0.15GB)	53.71% (19.71GB) +1.82% (+1.22GB)	0.60% (0.22GB) -0.02% (+0.00GB)

the communication cost profiling results outputted by HawkEye and SecretFLow-SEMI2K, the two-party backend of SecretFlow-SPU [30]. We use classical VGG-16 and ResNet-50 models as our target models. To obtain the dynamic communication cost profiling results, we run secure VGG-16 and ResNet-50 model training processes on SecretFLow-SEMI2K [6]. For parameter setting, we set the bit length as 64 and the bit length of fixed numbers' fractional part as 18. We set the statistical security parameter and computation security parameter of these two MPL frameworks as 40 and 128, respectively. Meanwhile, We use SGD as the optimizer. The input data size is $128 \times 3 \times 32 \times 32$, where 128 is the batch size.

To obtain the static model communication cost profiling results from HawkEye, we configure the model communication cost of SecretFLow-SEMI2K in HawkEye and implement the above models. Finally, we run HawkEye under the same parameter setting with SecretFLow-SEMI2K to obtain the static profiling results. We show the communication cost configuration of SecretFLow-SEMI2K in the full version of this manuscript.

Experimental results. As is shown in Table 9, HawkEye can accurately profile the communication cost of the secure model training process. The proportions of model training phases' online communication size to the total online communication size outputted by SecretFlow-SEMI2K and HawkEye are very close. Concretely, the proportion differences between baselines and HawkEye are all smaller than 1.82%. The main difference lies in the model backward process. The potential reason behind it is that SecretFlow-SEMI2K has optimizations for a few special circuit structures, such as consecutive multiplications used in the model backward process. Therefore, the online communication sizes of the model backward process outputted by SecretFlow-SEMI2K are slightly smaller than those outputted by HawkEye.

[6]The code repository address of SecretFLow-SEMI2K is https://github.com/secretflow/spu. We use SecretFlow-SPU0.9.3b0