From Individual Computation to Allied Optimization: Remodeling Privacy-Preserving Neural Inference with Function Input Tuning

Qiao Zhang[†], Tao Xiang[†*], Chunsheng Xin[‡], and Hongyi Wu[♮]
[†]*College of Computer Science, Chongqing University, Chongqing, 400044, China*
[‡]*Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA, 23529, USA*
[♮]*Department of Electrical and Computer Engineering, The University of Arizona, Tucson, AZ, 85721, USA*
E-mails: qiaozhang@cqu.edu.cn, txiang@cqu.edu.cn, cxin@odu.edu, mhwu@arizona.edu

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00101

***Abstract*—Privacy-preserving Machine Learning as a Service (MLaaS) enables the resource-limited client to cost-efficiently obtain inference output of a well-trained neural model that is possessed by the cloud server, with both client's input and server's model parameters protected. While efficiency plays a core role for practical implementation of privacy-preserving MLaaS and it is encouraging to witness recent advances towards efficiency improvement, there still exists a significant performance gap to real-world applications. The basic logic in state-of-the-art frameworks involves an individual computation for each function of the neural model, based on specific cryptographic primitives. While it is definitely logical, we look back to the necessity of this function-wise methodology and initiate the comprehensive exploration towards allied optimization for efficient privacy-preserving MLaaS. Under such fresh perspective, we remodel the computation process that is always from input to output of the same function in mainstream works, to the allied counterpart that is from one function's input associated with the start of expensive overhead to another function's output enabling effective circumvention of unnecessary cost within the procedure. As such we propose FIT (Function Input Tuning) which features by a computation module for composite function with a series of joint optimization strategies. Theoretically, FIT not only eliminates the most expensive crypto operations without invoking extra encryption enabler, but also makes the running-time crypto complexity independent of filter size. Experimentally, FIT demonstrates tens of times speedup over various function dimensions from modern models, and 4.5× to 35.5× speedup on the total computation time when plugged in neural networks with data from small-scale MNIST to large-scale ImageNet.**

1. Introduction

Deep Learning (DL) has undergone a remarkable evolution in the past decade [1], [2], [3], [4] and has proven to be highly effective in various smart services such as image classification [5], voice recognition [6], and financial evaluation [7]. However, the extensive demand for substantial training data and powerful computational resources [8] often renders DL impractical for end users with limited resources who want to train and apply DL models for their specific needs. To this end, Machine Learning as a Service (MLaaS) has emerged as a viable solution to alleviate such limitations. MLaaS establishes a service mode where a cloud server $\mathcal{S}$ owns a neural network that is well trained on plenty of data, and the client $\mathcal{C}$ uploads her input to $\mathcal{S}$ which runs the neural network and returns inference output to $\mathcal{C}$.

However, the issue of data privacy presents a significant challenge in the practical implementation of MLaaS, as it requires the client to transmit her private data, which may include sensitive information, to the server. This challenge arises from two main factors. Firstly, clients naturally desire to obtain the output of the MLaaS model without revealing their input to any other parties, including the server $\mathcal{S}$. In fact, there are regulations in place that prohibit the disclosure of private data in various domains, e.g., the Health Insurance Portability and Accountability Act (HIPAA) [9] for medical information and the General Data Protection Regulation (GDPR) [10] for business records. Secondly, the server $\mathcal{S}$ itself has an interest in protecting the proprietary parameters of its well-trained neural network. It seeks to maintain the confidentiality of these parameters and aims to only provide the model output in response to the client's inference query. By keeping the parameters hidden, the server can safeguard its intellectual property and prevent unauthorized access to its trained model.

Privacy-preserving MLaaS takes into account both legal and ethical concerns regarding the data privacy of both the client and the server. Its primary goal is to ensure that 1) the server remains oblivious to the client's private input and 2) the client does not gain access to the proprietary model parameters of the server beyond the necessary inference output, e.g., the predicted class. While privacy-preserving MLaaS offers a promising approach to reconcile MLaaS with data protection, it also presents a key challenge that must be addressed, i.e., how to efficiently embed cryptographic primitives into function computation of neural networks, which otherwise may lead to prohibitively high computation complexity and/or degraded prediction accuracy due to large-size circuits and/or function approximations.

To achieve usable privacy-preserving MLaaS, a series of recent works have made inspiring progress towards im-

*Corresponding author.

DOI 10.1109/SP54263.2024.00101

proving system efficiency [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27]. Notably, the inference speed has witnessed substantial improvement, with several orders of magnitude increase from early frameworks like CryptoNets [11] to more recent approaches such as CrypTFlow2 [17] and Cheetah [27]. At a high level, these privacy-preserving frameworks achieve greater computational efficiency by carefully applying suitable cryptographic primitives to calculate the linear (e.g., dot product and convolution) and nonlinear (e.g. ReLU) functions in a neural network. Among the cryptographic primitives, commonly used ones in these frameworks are Homomorphic Encryption (HE) [28], [29], [30], Oblivious Transfer (OT) [31], Secret Sharing (SS) [32] and Garbled Circuits (GC) [33], [34]. The mixed-primitive frameworks which utilize HE to compute linear functions while adapting Multi-Party Computation (MPC), such as GC and OT, for nonlinear functions have demonstrated additional efficiency advantages [12], [13], [17], [27]. This work makes a further step towards efficiency optimization of mixed-primitive frameworks.

In state-of-the-art privacy-preserving frameworks, particularly mixed-primitive frameworks, the core logic revolves around the fundamental concept of *individual computation for each function*. It first computes the output of a function, based on specific cryptographic primitives. Such securely computed output is in an "encrypted" form and is then shared between $\mathcal{C}$ and $\mathcal{S}$. The respective shares held by $\mathcal{C}$ and $\mathcal{S}$ then serve as the input of next function. As a neural network consists of stacked linear and nonlinear functions, this individual computation is sequentially applied on functions from beginning to the end. For example, the initial input of CrypTFlow2 [17] is $\mathcal{C}$'s private data, which is encrypted and sent to $\mathcal{S}$. $\mathcal{S}$ conducts HE-based computation for the linear function where the HE addition, multiplication, and rotation (which are three basic operators over encrypted data) are performed between $\mathcal{C}$-encrypted data and model parameters of $\mathcal{S}$. It produces an encrypted function output which is then shared between $\mathcal{C}$ and $\mathcal{S}$ in plaintext, and those shares serve as the input of the following OT-based computation, such as most significant bit (MSB) and multiplexing, for subsequent nonlinear function. Once again, the output shares from this step act as the input for the next function. This computation pattern continues in a function-wise manner until $\mathcal{C}$ obtains the final model output. Figure 1(a) shows the basic diagram of this process.

Key observations. While the individual computation for each function is a logical approach followed by most state-of-the-art privacy-preserving frameworks, there is a need to reconsider the computational cost associated with calculating that "encrypted" function output, particularly the rotations involved. The need to first compute such "encrypted" function output introduces *all* rotations, which are the most expensive HE operation among the three basic ones (addition, multiplication, and rotation). Minimizing the number of rotations becomes crucial in improving the overall computational efficiency of privacy-preserving frameworks [13], [17], [18], [27], [35]. However, the cost of HE-based operations in linear functions remains a bottleneck compared to the MPC-based operations in nonlinear functions. For instance, our preliminary experiments show that the HE-based convolution in VGG-16 [3] with input size $28 \times 28@512$, kernel size $3 \times 3@512$, and stride size 1×1 takes about 153 seconds under CrypTFlow2 framework on the Intel Core i7-11370H@3.30GHz CPU. In contrast, computing the OT-based nonlinear ReLU with the same input, which is actually the previous function to that convolution, only takes about 3.6 seconds, representing only around 2.3% of the time taken by the HE-based convolution. As a neural network consists of a stack of linear and nonlinear functions, this disparity highlights HE operations as the main obstacle in achieving efficient model computation for practical privacy-preserving MLaaS. Therefore, we are motivated to *totally eliminate* the involved rotations while *minimize the amount of HE multiplication and addition*, expecting a significant boost in the overall computation efficiency.

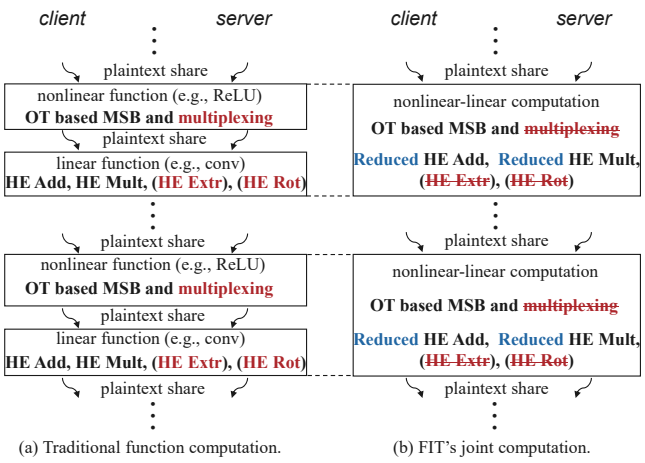

Figure 1. Comparison between mainstream and FIT.

In our exploration, we intend to consider efficiency optimization from the input of linear functions, which marks the beginning of all HE operations. Specifically, we investigate the convolution function denoted as $f_{\mathsf{c}}(y)$ and its nonlinear ReLU input denoted as $y = f_{\mathsf{r}}(x)$, which form a basic combination in modern networks [2], [3], [4]. Instead of minimizing the MPC-based cost for calculating $f_{\mathsf{r}}(x)$ with respect to x and the HE cost for computing $f_{\mathsf{c}}(y)$ with respect to input y, which is the mainstream logic in state-of-the-art schemes, we remodel such individual computation for each function to the allied procedure of composite function $f_{\mathsf{c}}(f_{\mathsf{r}}(x))$ with respect to input x. With this shift in perspective, we can potentially break the limitations of independently optimizing the efficiency of each function by focusing on the optimization of combined function. To make the computation of this composite function truly advantageous, we need to determine the right terms for each party to calculate, ensuring that the overall computation load for $f_{\mathsf{c}}(f_{\mathsf{r}}(x))$ is significantly reduced.

Given the input-independent pregeneration of one share

TABLE 1. RUNNING-TIME COMPLEXITIES COMPARED WITH STATE-OF-THE-ART PROTOCOLS FOR THE LINEAR AND NONLINEAR FUNCTIONS IN NEURAL NETWORKS. THE LINEAR FUNCTIONS INCLUDE CONVOLUTION (CONV) AND FULLY CONNECTION (FC), WHILE THE NONLINEAR FUNCTION IS RELU. THE CONV TAKES AS INPUT A 3-DIMENSION MATRIX WITH SIZE $C_i \times H_i \times W_i$. C_o AND f_h ARE THE NUMBER AND THE SIZE OF FILTERS, RESPECTIVELY. H_o AND W_o ARE THE HEIGHT AND WIDTH OF EACH 2-DIMENSION CONVOLUTION OUTPUT. THE FULLY CONNECTION TAKES AS INPUT A VECTOR WITH SIZE n_i AND OUTPUTS A VECTOR WITH SIZE n_o. THE NONLINEAR FUNCTION ACTS AS THE INPUT OF CONV OR FC, WITH DATA $\boldsymbol{x}$ IN SIZE $C_i \times H_i \times W_i$. rd IS THE NUMBER OF COMMUNICATION ROUNDS INVOLVED IN COMPUTATION FOR TARGET FUNCTIONS. $f'_{\mathsf{r}}(\boldsymbol{x})$ AND $\mathsf{Mx}(\boldsymbol{x})$ ARE THE DERIVATIVE OF RELU WITH INPUT $\boldsymbol{x}$ AND THE MULTIPLEXING TO PERFORM MULTIPLICATION BETWEEN $f'_{\mathsf{r}}(\boldsymbol{x})$ AND $\boldsymbol{x}$, RESPECTIVELY. THE COMMUNICATION COST IN ALL SCHEMES EXCLUDES COMMON OVERHEAD FOR COMPUTING $f'_{\mathsf{r}}(\boldsymbol{x})$.

Functions	Schemes	Computation Cost for HE Operations			Communication Cost	
		# Rot	# Extr	# Mult (Add)	# Ciphertexts	# Round
ReLU+Conv	DELPHI [16]	$O(C_iC_oH_iW_i(f_h)^2)$	-	$O(C_iC_oH_iW_i(f_h)^2)$	$O(C_iH_iW_i + C_oH_oW_o)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x}),\mathsf{Mx}(\boldsymbol{x})\}}$
	CrypTFlow2 [17]	$O(C_iC_oH_iW_i(f_h)^2)$	-	$O(C_iC_oH_iW_i(f_h)^2)$	$O(C_iH_iW_i + C_oH_oW_o)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x}),\mathsf{Mx}(\boldsymbol{x})\}}$
	Cheetah [27]	0	$O(C_oH_oW_o)$	$O(C_iC_oH_iW_i)$	$O(C_iH_iW_i + C_oH_oW_o)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x}),\mathsf{Mx}(\boldsymbol{x})\}}$
	FIT	0	0	$O(C_iH_iW_i)$	$O(C_iH_iW_i)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x})\}}$
ReLU+FC	HElib [36]	$O(n_i)$	-	$O(n_i)$	$O(n_i)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x}),\mathsf{Mx}(\boldsymbol{x})\}}$
	GAZELLE [13]	$O(n_in_o - \log n_o)$	-	$O(n_in_o)$	$O(n_in_o)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x}),\mathsf{Mx}(\boldsymbol{x})\}}$
	Cheetah [27]	0	$O(n_o)$	$O(n_in_o)$	$O(n_in_o)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x}),\mathsf{Mx}(\boldsymbol{x})\}}$
	FIT	0	0	$O(n_i)$	$O(n_i)$	$1+\mathsf{rd}_{\{f'_{\mathsf{r}}(\boldsymbol{x})\}}$

of each function output, we design an online-offline assignment of unfolded terms in $f_{\mathsf{c}}(f_{\mathsf{r}}(x))$, which results in *a total elimination of all rotations and a filter-independent computation complexity of HE operations at running time.* The basic idea is to identify, utilize and construct a set of pre-generated shares to form specific ciphertext and masked intermediates in a client-data-independent offline phase, while making the data-dependent online computation not only escape part of MPC-based computation such as multiplexing, which is not possible under mainstream individual computation, but also merely involve a small number of HE addition and multiplication. We call such methodology as Function Input Tuning (FIT) which tunes the unfolded terms in composite function to form new terms that serve as the underlying input, with computation efficiency highly improved.

In contrast, the most recent Cheetah [27] removes rotation by introducing a large amount of HE extraction (Extr) and its complexity for HE addition and multiplication is proportional to the number of output channels namely C_o from filter size. While Zhang et al. [35] escapes rotation through an intensive ciphertext communication at running time, which requires a scrupulous balance between bandwidth and overall efficiency. The allied optimization for composite function with filter-independent crypto complexity at running time makes FIT distinguishable among current works for efficient privacy-preserving MLaaS. For example, computing adjacent ReLU and convolution with aforementioned sizes takes about 4.5 seconds by FIT, indicating a speedup about 35 times compared to conventional approaches. A high-level idea of the proposed FIT is illustrated in Figure 1(b).

Our contributions. Overall, the contributions of this work are summarized as follows.

- We embark on a comprehensive exploration of efficient computation for composite function and introduce our approach called FIT. FIT challenges the individual computation for each function that is followed by mainstream works and remodels the function-wise process by an allied procedure. As such, FIT devises a computation module for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ with lightweight HE complexity at running time, incorporating a series of joint optimization strategies.
- The efficiency advantages of FIT are highlighted in Table 1. One notable benefit is that FIT eliminates rotations without HE extraction, and the running-time complexity of HE multiplication and addition is independent of filter size namely f_h and C_o. Such reduction in complexity contributes to the overall efficiency and effectiveness of FIT when integrated into neural models.
- Through extensive experiments detailed in Section 4, FIT surpasses the efficiency of function-wise computation typically employed in the state-of-the-art frameworks. The results showcase tens of times speedup achieved by FIT across varying function dimensions from modern neural models. Furthermore, when integrated into neural networks with datasets from small-scale MNIST to large-scale ImageNet, FIT achieves notable speed gains of 4.5× to 35.5×. These experimental findings firmly establish the superiority of FIT in terms of computational efficiency and overall performance.

The rest of the paper is organized as follows. In Section 2, we introduce system setup and primitives that are adopted in FIT. Section 3 elaborates the design of FIT for computing $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ as well as the strategies that best adapt FIT to neural networks. The experimental results are illustrated and discussed in Section 4. Finally, we conclude the paper in Section 5.

2. Preliminaries

Notations. We denote $[\![i]\!] = \{0, 1, \ldots, i-1\}$ for $i \in \mathbb{N}$. $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ are the ceiling and flooring functions, respectively. $\mathbf{1}\{\mathcal{I}\}$ is the indicator function that is 1 when $\mathcal{I}$ is true and 0 when $\mathcal{I}$ is false. $r \xleftarrow{\$} \mathcal{D}$ randomly samples a component r from a set $\mathcal{D}$. The logical XOR is denoted as $\oplus$. $\mathbb{Z}_p = \mathbb{Z} \cap [-\lfloor p/2 \rfloor, \lfloor p/2 \rfloor]$ for $p > 2$, and $\mathbb{Z}_2 = \{0, 1\}$.

Furthermore, $+, -$, and $\boxtimes$ are element-wise addition, subtraction, and multiplication, respectively, in either ciphertext or plaintext domain depending on whether ciphertext is involved or not.

2.1. System Model

We consider the context of cryptographic inference as shown in Figure 2 where $\mathcal{C}$ holds a private input and $\mathcal{S}$ holds the neural network with proprietary model parameters. After the inference, $\mathcal{C}$ learns two pieces of information: the network architecture (such as the number, types and dimensions of involved functions) and the network output, while $\mathcal{S}$ learns nothing. These learnt information is commonly assumed in state-of-the-art frameworks such as Cheetah [27]. Such scenario can be applied in assistant decision-making system for clinical diagnosis, which has been deployed by Ant Group. For example, a small clinic could obtain more precise diagnostic analysis result from central hospitals that possess more comprehensive DL models.

To complete such cryptographic inference, the server processes $\mathcal{C}$'s input through a sequence of linear and nonlinear functions of its neural network to finally classify that input into one of the potential classes. Specifically, we target at the widely-applied Convolutional Neural Network (CNN) and describe in the following its functions that are mainly investigated in this paper.

Linear functions. ① Convolution (Conv). The Conv $f_{\mathsf{c}}(\cdot)$ operates between an input $\boldsymbol{a} \in \mathbb{Z}_p^{C_i \times H_i \times W_i}$ and kernel $\boldsymbol{k} \in \mathbb{Z}_p^{C_o \times C_i \times f_h \times f_h}$ with stride $s \in \mathbb{N}^+$. Here C_i and C_o are the number of input and output channels, respectively. H_i and W_i are the height and width of each two-dimension input channel, and f_h is the size of each two-dimension filter of kernel $\boldsymbol{k}$. The output includes C_o channels, each of which is derived from computation between $\boldsymbol{a}$ and one of C_o filters in $\boldsymbol{k}$. Specifically, a value in one output channel is the sum of all $C_i {f_h}^2$ elements in one filter, each of which is scaled by same-location value of $\boldsymbol{a}$ within the filter window. The next value in that output channel is similarly obtained by gradually moving the filter with stride s over $\boldsymbol{a}$. The summing process in Conv inevitably introduces a series of expensive operations, such as rotation, in cryptographic inference. In this paper, we take the previous function of Conv as its input and propose to jointly optimize computation efficiency of such composite function, which results in an efficient computation module as stated in Section 3.2.

② Fully Connection (FC). The input to FC $f_{\mathsf{w}}(\cdot)$ is an n_i-sized vector $\boldsymbol{a} \in \mathbb{Z}_p^{n_i}$ and a weight matrix $\boldsymbol{w} \in \mathbb{Z}_p^{n_o \times n_i}$. The output is one n_o-sized vector where each value is the sum of elements in vector obtained by multiplying one row of $\boldsymbol{w}$ with $\boldsymbol{a}$. Similar with Conv, we take the previous function of $f_{\mathsf{w}}(\cdot)$ into account and propose an efficient computation module for such composite function as stated in Section 3.2. ③ Batch Normalization (BN). In the inference of a neural network, BN $f_{\mathsf{bn}}(\cdot)$ scales and shifts each two-dimension input channel by a constant, respectively. Here, the input $\boldsymbol{a} \in \mathbb{Z}_p^{C_i \times H_i \times W_i}$. As BN always follows behind Conv, we integrate it with Conv to facilitate the use of our efficient computation module for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$. The detail of such integration is presented in Section 3.3. ④ Mean Pooling (MeanPool). Given the input $\boldsymbol{a} \in \mathbb{Z}_p^{C_i \times H_i \times W_i}$, MeanPool $f_{\mathsf{mn}}(\cdot)$ sums and averages components of $\boldsymbol{a}$ in each s_{n}-by-s_{n} pooling window where $s_{\mathsf{n}} \in \mathbb{N}^+$, and all averaged values form the final output. In this way, the output size becomes $C_i \times \left\lceil \frac{H_i}{s_{\mathsf{n}}} \right\rceil \times \left\lceil \frac{W_i}{s_{\mathsf{n}}} \right\rceil$.

Nonlinear functions. ① ReLU. For a value $a \in \mathbb{Z}_p$, the ReLU $f_{\mathsf{r}}(\cdot)$ is calculated as $f_{\mathsf{r}}(a) = a \boxtimes \mathbf{1}\{a\}$. Since ReLU always serves as input of Conv and FC in a neural network, we combine ReLU with Conv and FC, and aim to jointly optimize the computation efficiency of $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ and $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$. ② Max Pooling (MaxPool). the MaxPool $f_{\mathsf{mx}}(\cdot)$ works similarly with MeanPool except that every returned value is the maximum rather than the mean in each pooling window. Since MeanPool and MaxPool always appear between Conv and ReLU, we make them merged with adjacent functions, as described in Section 3.3, to enable the utilization our joint computation module for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ to realize a more efficient cryptographic inference.

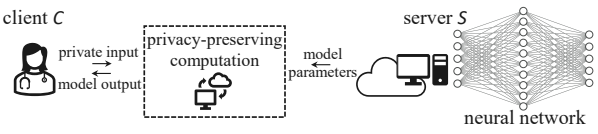

Figure 2. Overview of Cryptographic Inference.

2.2. Threat Model and Security

We prove security against a semi-honest adversary in Section 3.4 under the simulation paradigm [37]. Specifically, a computationally bounded adversary corrupts either $\mathcal{C}$ or $\mathcal{S}$ at the beginning of the protocol while it follows the protocol specification honestly. Security is modeled by defining two interactions: a real-world interaction where the client and the server execute protocol in presence of an adversary and environment, and an ideal-world interaction where both parties send their inputs to a trusted third party that computes target functionality faithfully. Security requires that for every adversary in real world, there is a simulator, in the ideal world, which makes no environment be able to distinguish between real-world and ideal-world interactions.

2.3. Cryptographic Primitives

We mainly rely on two cryptographic primitives in this paper to develop FIT. The concrete description is as follows.

Homomorphic Encryption (HE). HE is a primitive that supports various vector operations over encrypted data without decryption. It produces an encrypted output which matches the corresponding operations on plaintext [13]. Specifically, given a vector $\boldsymbol{x} = (x_0, x_1, \ldots, x_{N-1}) \in \mathbb{Z}_p^N$, it is encrypted into a ciphertext $[\boldsymbol{x}] = E(pk, \boldsymbol{x})$ where pk is the public key of either the client $\mathcal{C}$ or the server $\mathcal{S}$. The correctness of HE is firstly guaranteed by a decryption

process such that $\boldsymbol{x} = D(sk, [\boldsymbol{x}])$ where sk is the secret key of either $\mathcal{C}$ or $\mathcal{S}$. Second, the HE system is able to securely evaluate an arithmetic circuit consisting of addition and multiplication gates by leveraging the following operations. 1) Homomorphic addition $(+)$: $D(sk, [\boldsymbol{u}] + [\boldsymbol{v}]) = \boldsymbol{u} + \boldsymbol{v}$ and $D(sk, [\boldsymbol{u}] + \boldsymbol{v}) = \boldsymbol{u} + \boldsymbol{v}$. 2) Homomorphic subtraction $(-)$: $D(sk, [\boldsymbol{u}] - [\boldsymbol{v}]) = \boldsymbol{u} - \boldsymbol{v}$ and $D(sk, [\boldsymbol{u}] - \boldsymbol{v}) = \boldsymbol{u} - \boldsymbol{v}$. 3) Plaintext multiplication $(\boxtimes)$: $D(sk, [\boldsymbol{u}] \boxtimes \boldsymbol{v}) = \boldsymbol{u} \boxtimes \boldsymbol{v}$. 4) Homomorphic rotation (Rot): $D(sk, R([\boldsymbol{u}], l)) = \boldsymbol{u}_\ell$ where $\boldsymbol{u}_\ell = (u_\ell, \ldots, u_{N-1}, u_0, \ldots, u_{\ell-1})$ and $\ell \in [\![N]\!]$. Note that a rotation by $(-\ell)$ is the same as a rotation by $(N-\ell)$. Here $\boldsymbol{u} = (u_0, \ldots, u_{N-1}) \in \mathbb{Z}_p^N$ and $\boldsymbol{v} = (v_0, \ldots, v_{N-1}) \in \mathbb{Z}_p^N$.

Given above four operations, the running-time complexity of Rot is significantly larger than that of others [13], [24]. Specifically, one Rot costs as much as performing a Number Theoretic Transform (NTT) and a large number of inverse NTTs which is proportional to the bitlength of ciphertext modulus. While NTT and inverse NTT are computationally expensive, with which other HE operations does not need to involve, it makes Rot the most expensive one among HE operations. Therefore, optimizing the complexity of HE operations, especially Rot, with respect to $f_{\mathsf{c}}(\boldsymbol{x})$ and $f_{\mathsf{w}}(\boldsymbol{x})$ is one of the mainstream solutions in cryptographic inference to reduce computation cost for Conv and FC [13], [27]. Different from current works that focus on independent efficiency optimization for functions $f_{\mathsf{c}}(\boldsymbol{x})$ and $f_{\mathsf{w}}(\boldsymbol{x})$, we take their previous functions into consideration to jointly optimize crypto cost of $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ and $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$. This shift in perspective results in our efficient module as elaborated in Section 3.2.

Additive Secret Sharing (ASS). ASS generates shares of input $x \in \mathbb{Z}_p$ as shr_0 and shr_1. Such shares are randomly sampled in $\mathbb{Z}_p$ while satisfy $shr_0 + shr_1 = x \bmod p$. By sharing specifically-crafted intermediates in the process of joint computation, we efficiently eliminate all Rot in the cryptographic inference, with much less complexities of HE multiplication and addition at running-time.

3. System Description

3.1. Overview

Figure 3 shows the overview of FIT's methodology to compute composite function $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$. Instead of optimize the computation efficiency in a function-wise manner, which is the widely-adopted logic in mainstream works, FIT remodels such procedure into an allied counterpart from one function's input that is associated with the start of expensive operations to another function's output that enables a much efficient computation within the whole composite processing. We begin such exploration by unfolding the composite $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ where the $f_{\mathsf{r}}(\boldsymbol{x})$ is the start of expensive HE operations. Our key idea is to decompose that composite function into the sum of multipliers, each of which is expected to be the multiplication between term that contains shares from one party and single term that can be pregenerated from the same or the other party. As such, by taking advantages of the share pregeneration before the prediction query, we could possibly eliminate computation for part or the whole of decomposed multipliers at running time. Under such guidance, we divide the whole process of getting $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ into two phases namely offline computation and online computation. The offline computation involves pregenerating specifically crafted ciphertext as well as sharing the privately computed secret, all based on pregenerated terms from either the client and the server. While the online computation includes an OT-based module to obtain the client's share of derivative, and an one-round interaction to finally obtain the share of composite function $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$.

Such specific design for offline and online derives from our further investigation to make both online computation and offline computation efficient. First, our offline computation eliminates the expensive rotations through a row-accumulated mechanism that makes full use of every slot in a packed ciphertext. Second, by taking advantages of the terms from offline computation, our online computation is completed in a noticeably efficient way that the crypto complexity is independent of kernel size and the communication round after OT-based module is reduced from three to only one. We elaborate the details of FIT's joint optimization for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ as well as FIT's complexity to calculate such composite functions in Section 3.2. Furthermore, a neural network always includes other functions besides composite function $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$, such as Maxpooling, Meanpooling and Batch Normalization, we introduce in Section 3.3 a series of model adjustments which equalize functions into another form such that $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ appears as much as possible to enable best utilization of FIT's computation modules. Finally, Section 3.4 analyzes and proves the security of FIT under semi-honest assumption.

3.2. Joint Optimization for $f_{\mathbf{c}}(f_{\mathbf{r}}(\boldsymbol{x}))$

Recall that we aim to jointly optimize composite function $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ such that the computation is much more efficient than first getting $\boldsymbol{y} = f_{\mathrm{r}}(\boldsymbol{x})$ and then obtaining $f_{\mathrm{c}}(\boldsymbol{y})$, which is the mainstream logic in state-of-the-art approaches. We start such exploration by looking at the unfolded expression of $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ as

$$f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x})) = \boldsymbol{k} * f_{\mathsf{r}}(\boldsymbol{x}) = \boldsymbol{k} * \{f'_{\mathsf{r}}(\boldsymbol{x}) \boxtimes \boldsymbol{x})\} \tag{1}$$

$$= \boldsymbol{k} * \{\boldsymbol{h}_1 + \boldsymbol{h}_2 \boxtimes \boldsymbol{g}_1(\boldsymbol{x}) + \boldsymbol{h}_3 \boxtimes \boldsymbol{g}_0(\boldsymbol{x})\} + \boldsymbol{k} * \boldsymbol{h}_4 \tag{2}$$

$$= \boldsymbol{k} * \boldsymbol{h}_5 + \boldsymbol{k} * \boldsymbol{h}_4 \tag{3}$$

where

$$\begin{cases} \boldsymbol{h}_1 = \boldsymbol{x}_0 \boxtimes \boldsymbol{g}_0(\boldsymbol{x}) \\ \boldsymbol{h}_2 = \boldsymbol{x}_0 \boxtimes \{1 - 2 \boxtimes \boldsymbol{g}_0(\boldsymbol{x})\} \\ \boldsymbol{h}_3 = \boldsymbol{x}_1 \boxtimes \{1 - 2 \boxtimes \boldsymbol{g}_1(\boldsymbol{x})\} \\ \boldsymbol{h}_4 = \boldsymbol{x}_1 \boxtimes \boldsymbol{g}_1(\boldsymbol{x}) \\ \boldsymbol{h}_5 = \boldsymbol{h}_1 + \boldsymbol{h}_2 \boxtimes \boldsymbol{g}_1(\boldsymbol{x}) + \boldsymbol{h}_3 \boxtimes \boldsymbol{g}_0(\boldsymbol{x}) \end{cases} \tag{4}$$

kernel $\boldsymbol{k} \in \mathbb{Z}_p^{C_o \times C_i \times f_h \times f_h}$, $\boldsymbol{x}, \boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{h}_1, \boldsymbol{h}_2, \boldsymbol{h}_3, \boldsymbol{h}_4 \in \mathbb{Z}_p^{C_i \times H_i \times W_i}$. $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ are shares at $\mathcal{C}$ and $\mathcal{S}$ satisfying $\boldsymbol{x}_0 + \boldsymbol{x}_1 = \boldsymbol{x}$, $\boldsymbol{g}_0(\boldsymbol{x})$ and $\boldsymbol{g}_1(\boldsymbol{x})$ are the respective Boolean

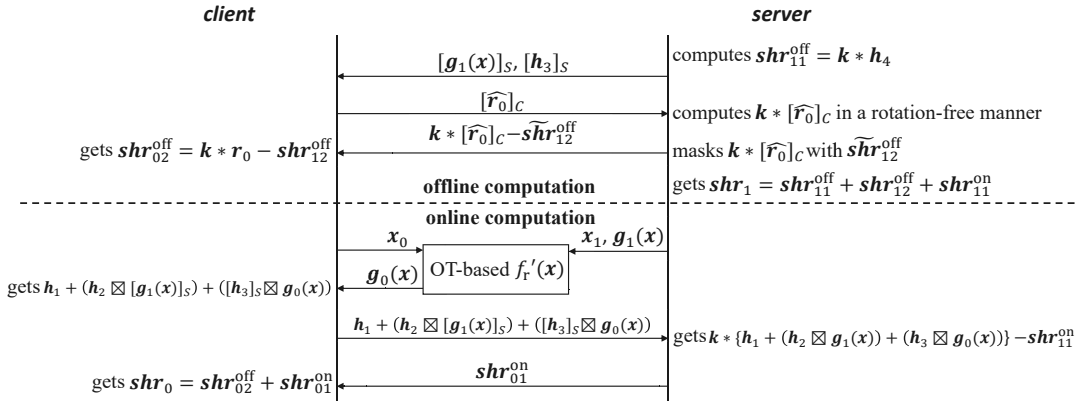

Figure 3. Overview of our proposed FIT to compute composite function $f_c(f_r(\boldsymbol{x}))$.

shares of client and server, obtained from the OT-based module for getting $f_r'(\boldsymbol{x})$ such that their XOR satisfies $\boldsymbol{g}_0(\boldsymbol{x}) \oplus \boldsymbol{g}_1(\boldsymbol{x}) = f_r'(\boldsymbol{x}) \in \mathbb{Z}_2^{C_i \times H_i \times W_i}$. Here we unfold $f_c(f_r(\boldsymbol{x}))$ by putting together the variables that belong to either $\mathcal{C}$ or $\mathcal{S}$.

Based on Eq. (3), we have the following observations and analyses by prioritizing cheap computation (e.g., local and plaintext calculation), which form the basic idea of FIT. First, since the $\boldsymbol{k}$ and $\boldsymbol{h}_4$ are at the server, $\mathcal{S}$ is able to obtain plaintext $\boldsymbol{k} * \boldsymbol{h}_4$ in Eq. (3). Second, the server could further compute $\boldsymbol{k} * \boldsymbol{h}_4$ independently if $\boldsymbol{x}_1$ and $\boldsymbol{g}_1(\boldsymbol{x})$ were pre-generated. Third, if $\mathcal{S}$ has a share of $\boldsymbol{h}_5$ namely $\boldsymbol{h}_5 - \boldsymbol{r}$, it could get $\boldsymbol{k} * (\boldsymbol{h}_5 - \boldsymbol{r})$ in plaintext. Fourth, the client and the server could get shares of $\boldsymbol{k} * \boldsymbol{r}$ in a input-independent phase, if $\boldsymbol{r}$ were pre-generated by $\mathcal{C}$ namely $\boldsymbol{r} = \boldsymbol{r}_0$. As for the first and second points, we make $\boldsymbol{x}_1$ and $\boldsymbol{g}_1(\boldsymbol{x})$ pre-generated by $\mathcal{S}$ such that we are able to get $\boldsymbol{k} * \boldsymbol{h}_4$ in an offline phase. It is doable since one of the shares of either $\boldsymbol{x}$ or $f_r'(\boldsymbol{x})$ could be pre-determined by one party. As for the third and fourth points, we reset $\boldsymbol{h}_1 = \{\boldsymbol{x}_0 \boxtimes \boldsymbol{g}_0(\boldsymbol{x}) - \boldsymbol{r}_0\}$ and modify Eq. (3) as

$$\boldsymbol{k} * \boldsymbol{h}_5 + \boldsymbol{k} * \boldsymbol{h}_4 + \boldsymbol{k} * \boldsymbol{r}_0 \tag{5}$$

where $\boldsymbol{r}_0$ is pre-determined by $\mathcal{C}$. In this way, the server is able to compute $\boldsymbol{k} * \boldsymbol{h}_5$ in plaintext since $\boldsymbol{h}_5$ is now masked by $\mathcal{C}$'s $\boldsymbol{r}_0$, in the reset $\boldsymbol{h}_1$. Meanwhile, such computation is completed in an online phase since $\boldsymbol{h}_5$ involves $\boldsymbol{x}_0$ and $\boldsymbol{g}_0(\boldsymbol{x})$. Therefore, the first thing we need to tackle is to make the server efficiently obtain $\boldsymbol{h}_5$. Furthermore, the data-independent nature of $\boldsymbol{r}_0$ makes $\boldsymbol{k} * \boldsymbol{r}_0$ to be computed in the offline phase, and thus the second thing we need to address is to efficiently compute $\boldsymbol{k} * \boldsymbol{r}_0$. Solving the above two problems would allow us efficiently obtain the shares of $f_c(f_r(\boldsymbol{x}))$ at $\mathcal{C}$ and $\mathcal{S}$.

We deal with above two issues through a carefully designed online-offline mechanism. By fully utilizing the pregeneration of $\boldsymbol{x}_1, \boldsymbol{g}_1(\boldsymbol{x}), \boldsymbol{r}_0$, and the OT-based module for getting shares of $f_r'(\boldsymbol{x})$, we are able to obtain shares of $f_c(f_r(\boldsymbol{x}))$ in a much more efficient way, under the joint optimization logic. Generally, the offline phase involves a rotation-free computation, with only HE addition and multiplication, to obtain shares of $\boldsymbol{k} * \boldsymbol{r}_0$. Given the $\mathcal{S}$-encrypted $\boldsymbol{g}_1(\boldsymbol{x})$ and $\boldsymbol{h}_3$, our online phase makes the client obtain encrypted $\boldsymbol{h}_5$, which is then decrypted by the server, in a non-multiplexing and rotation-free way, with the complexities of HE addition and multiplication independent of kernel sizes and only related to the number of input ciphertext. The details of our online-offline computation is elaborated as follows.

Rot-free sharing for $\boldsymbol{k} * \boldsymbol{r}_0$ at offline phase. Given that the kernel $\boldsymbol{k}$ is at $\mathcal{S}$ and $\boldsymbol{r}_0$ is at $\mathcal{C}$, state-of-the-art frameworks usually let the client encrypt $\boldsymbol{r}_0$ and the convolution is then performed at $\mathcal{S}$ through massive amount of HE operations with both rotations and multiplications in $O(C_i C_o H_i W_i (f_h)^2)$ [17], or trading rotations by HE extractions and decryptions both in $O(C_o H_o W_o)$ with multiplications in $O(C_i C_o H_i W_i)$ [27]. By comparison, we eliminate rotations by decryptions in $O(C_o)$ and multiplications in $O(C_i C_o H_i W_i (f_h)^2)$, without extractions. While f_h is always small (e.g., one or three), $H_o W_o$ could be hundreds or thousands or even larger in modern networks [3], [4]. Therefore, our calculation for $\boldsymbol{k} * \boldsymbol{r}_0$ provides a noticeable advantage over practical neural models by trading more HE decryptions for much less HE multiplications, besides its offline-computed nature as a time-saving remedy.

Specifically, our idea is based on the relationships among convolution, dot product, and HE operations, as illustrated in Figure 4. Since each number of convolution output is the sum of kernel values scaled by input elements that are within the kernel window, the overall convolution is equivalent to the dot product between a flattened kernel matrix and a re-organized input matrix [38]. More generally, the convolution between $\boldsymbol{k} \in \mathbb{Z}_p^{C_o \times C_i \times f_h \times f_h}$ and $\boldsymbol{r}_0 \in \mathbb{Z}_p^{C_i \times H_i \times W_i}$ could be viewed as the dot product between $\widehat{\boldsymbol{k}} \in \mathbb{Z}_p^{C_o \times C_i (f_h)^2}$ and $\widehat{\boldsymbol{r}_0} \in \mathbb{Z}_p^{C_i (f_h)^2 \times H_o W_o}$. For each of the C_o rows in $\widehat{\boldsymbol{k}}$, the dot product with $\widehat{\boldsymbol{r}_0}$ corresponds to one of the C_o H_o-by-W_o channels in convolution output, and that dot product is also the accumulation of all rows in $\widehat{\boldsymbol{r}_0}$ each of which is scaled by one value in the row of $\widehat{\boldsymbol{k}}$. For example, the convolution between $\boldsymbol{k} \in \mathbb{Z}_p^{1 \times 1 \times 3 \times 3}$ and $\boldsymbol{r}_0 \in \mathbb{Z}_p^{1 \times 2 \times 2}$ in Figure 4 is transformed into the dot product between $\widehat{\boldsymbol{k}} \in$

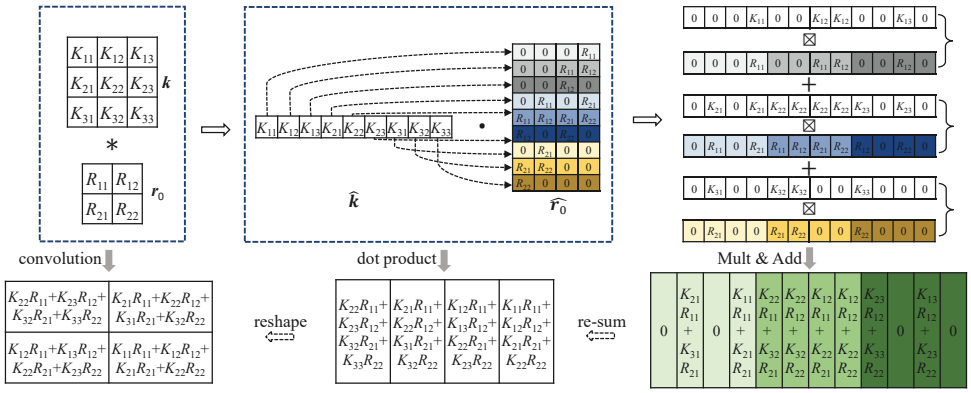

Figure 4. Relationships among convolution, dot product, and HE operations.

$\mathbb{Z}_p^{1\times 9}$ and $\widehat{\boldsymbol{r}_0} \in \mathbb{Z}_p^{9\times 4}$ with $H_o = W_o = W_i = 2$, and the dot product between first row of $\widehat{\boldsymbol{k}}$ and $\widehat{\boldsymbol{r}_0}$ is equivalent to the accumulation of all rows in $\widehat{\boldsymbol{r}_0}$, with j-th row scaled by j-th value in first row of $\widehat{\boldsymbol{k}}$.

On the other hand, the additive property of rows in $\widehat{\boldsymbol{r}_0}$ for getting each H_o-by-W_o channel of convolution output is compatible with the HE additions among ciphertext, since if we pack multiple rows of $\widehat{\boldsymbol{r}_0}$ into one ciphertext, multiply the totally $C_i(f_h)^2/C_n$ ciphertext with associated kernel values in a row of $\widehat{\boldsymbol{k}}$ by HE Mult where $C_n = \lfloor N/H_oW_o \rfloor$ is the number of rows that are packed in one ciphertext, and add these multiplied ciphertext together by HE Mult, we finally obtain a ciphertext which contains partial sums of one H_o-by-W_o channel of convolution output, and a re-summing of these decrypted values produces the desired convolution output. For example, three rows of $\widehat{\boldsymbol{r}_0}$ in Figure 4 are packed into one ciphertext and we thus obtain totally three ciphertext, which are multiplied with kernel values of first row in $\widehat{\boldsymbol{k}}$ by HE Mult, and then added up by HE Add. The resultant ciphertext contains partial sums of the convolution output, and a re-summing of the decrypted values turns out to be desired convolution output.

The above methodology applies to dot product between each row of $\widehat{\boldsymbol{k}}$ and $\widehat{\boldsymbol{r}_0}$, which forms FIT's main process to share $\boldsymbol{k} * \boldsymbol{r}_0$ in an efficient and privacy-preserving way: the client encrypts rows of $\widehat{\boldsymbol{r}_0}$ and forms totally $C_i(f_h)^2/C_n$ ciphertext, which are sent to the server. For each row of $\widehat{\boldsymbol{k}}$, $\mathcal{S}$ multiplies each of the $C_i(f_h)^2/C_n$ ciphertext with values in that row of $\widehat{\boldsymbol{k}}$, and adds the multiplied ciphertext to form an intermediate one. The resultant C_o intermediate ciphertext are masked by $\widetilde{\boldsymbol{shr}}_{12}^{\text{off}}$, which is re-summed into $\boldsymbol{shr}_{12}^{\text{off}}$, and sent to the client, which performs the decryption and gets the share of $\boldsymbol{k}*\boldsymbol{r}_0$ namely $\boldsymbol{shr}_{02}^{\text{off}}$. The protocol of our Rot-free sharing for $\boldsymbol{k}*\boldsymbol{r}_0$ at offline phase is described in Figure 5.

Additionally, there involves a unidirectional offline transmission from $\mathcal{S}$ to $\mathcal{C}$ where the server respectively encrypts $\boldsymbol{g}_1(\boldsymbol{x})$ and $\boldsymbol{h}_3$ into C_i/C_n ciphertext as $[\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}$ and $[\boldsymbol{h}_3]_{\mathcal{S}}$, and sends them to the client. The sharing for $\boldsymbol{k}*\boldsymbol{r}_0$ together with the unidirectional offline transmission facilitate FIT's efficient online computation to be discussed next.

Input:
$\mathcal{S}$: $\boldsymbol{k} \in \mathbb{Z}_p^{C_o\times C_i\times f_h\times f_h}$
$\mathcal{C}$: $\boldsymbol{r}_0 \in \mathbb{Z}_p^{C_i\times H_i\times W_i}$
Output:
$\mathcal{S}$: a random share $\boldsymbol{shr}_{12}^{\text{off}} \in \mathbb{Z}_p^{C_o\times H_o\times W_o}$;
$\mathcal{C}$: $\boldsymbol{shr}_{02}^{\text{off}} \in \mathbb{Z}_p^{C_o\times H_o\times W_o}$, s.t., $\boldsymbol{shr}_{12}^{\text{off}} + \boldsymbol{shr}_{02}^{\text{off}} \pmod p = \boldsymbol{k} * \boldsymbol{r}_0$.

$\mathcal{C}$:	$\mathcal{S}$:
$\widehat{\boldsymbol{r}_0} \in \mathbb{Z}_p^{C_i(f_h)^2\times H_oW_o} \leftarrow \boldsymbol{r}_0$	$\widehat{\boldsymbol{k}} \in \mathbb{Z}_p^{C_o\times C_i(f_h)^2} \leftarrow \boldsymbol{k}$
$[\widehat{\boldsymbol{r}_0}]_C \leftarrow E(pk_C, \widehat{\boldsymbol{r}_0})$	$\widetilde{\boldsymbol{shr}}_{12}^{\text{off}} \xleftarrow{\$} \mathbb{Z}_p^{C_o\times N}$
	$[\widehat{\boldsymbol{r}_0}]_C \rightarrow$
	$\widetilde{\boldsymbol{shr}}_{02}^{\text{off}} \leftarrow \sum([\widehat{\boldsymbol{r}_0}]_C \boxtimes \widehat{\boldsymbol{k}}) - \widetilde{\boldsymbol{shr}}_{12}^{\text{off}}$
$\leftarrow \widetilde{\boldsymbol{shr}}_{02}^{\text{off}}$	
$\boldsymbol{shr}_{02}^{\text{off}} \leftarrow \sum(D(sk_C, \widetilde{\boldsymbol{shr}}_{02}^{\text{off}}))$	$\boldsymbol{shr}_{12}^{\text{off}} \leftarrow \sum\widetilde{\boldsymbol{shr}}_{12}^{\text{off}}$
Output $\boldsymbol{shr}_{02}^{\text{off}}$	Output $\boldsymbol{shr}_{12}^{\text{off}}$

Figure 5. Rot-free sharing for $\boldsymbol{k} * \boldsymbol{r}_0$ at offline phase.

Rot-free computation for $\boldsymbol{h}_5$ at online phase. Recall that we aim to privately get the shares of $f_c(f_r(\boldsymbol{x}))$ in a more efficient way than sequentially computing $\boldsymbol{y} = f_r(\boldsymbol{x})$ and $f_c(\boldsymbol{y})$. We equalize the problem to Eq. 5 and break our goal into two concrete tasks: sharing of $\boldsymbol{k} * \boldsymbol{r}_0$ and $\mathcal{S}$'s acquisition of $\boldsymbol{h}_5$. While $\boldsymbol{k} * \boldsymbol{h}_4$ in Eq. 5 is locally obtained by $\mathcal{S}$ and $\boldsymbol{k}*\boldsymbol{r}_0$ is shared according to FIT's offline mechanism described before, our remaining task is to make $\mathcal{S}$ efficiently obtain $\boldsymbol{h}_5$ such that $\boldsymbol{k} * \boldsymbol{h}_5$ is computed and the shares of $f_c(f_r(\boldsymbol{x}))$ are thus formed. We achieve a light-weight computation for $\boldsymbol{h}_5$ by making use of the OT-based online sharing for $f'_r(\boldsymbol{x})$ and our offline computation module. Specifically, given the shares of $\boldsymbol{x}$ namely $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, $\mathcal{C}$ obtains the Boolean share of $f'_r(\boldsymbol{x})$ namely $\boldsymbol{g}_0(\boldsymbol{x})$ based on the optimized OT-based protocol for $f'_r(\boldsymbol{x})$ [17], while $\mathcal{S}$'s corresponding share $\boldsymbol{g}_1(\boldsymbol{x})$ is pregenerated. Considering the $[\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}$ and $[\boldsymbol{h}_3]_{\mathcal{S}}$ that are received by $\mathcal{C}$ at offline phase, the client is able to compute an encrypted $[\boldsymbol{h}_5]_{\mathcal{S}}$ right after it gets $\boldsymbol{g}_0(\boldsymbol{x})$ as

$$[\boldsymbol{h}_5]_{\mathcal{S}} = \boldsymbol{h}_1 + (\boldsymbol{h}_2 \boxtimes [\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}) + ([\boldsymbol{h}_3]_{\mathcal{S}} \boxtimes \boldsymbol{g}_0(\boldsymbol{x})) \quad (6)$$

where there involve $2C_i/C_n$ HE Mult and $2C_i/C_n$ HE Add, and $[\boldsymbol{h}_5]_{\mathcal{S}}$ obviously includes C_i/C_n ciphertext since both

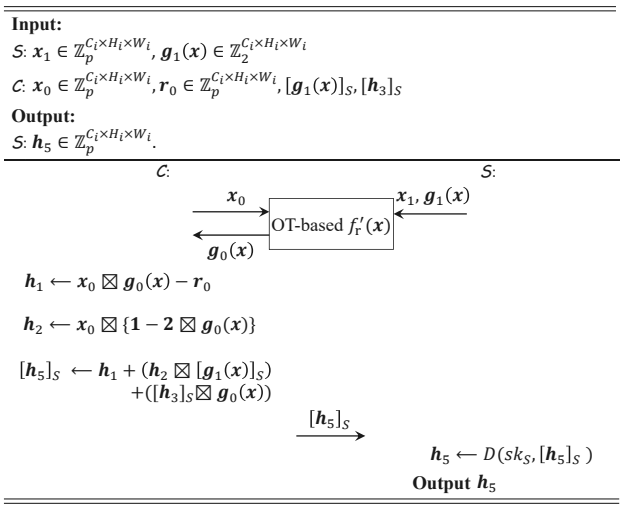

Figure 6. Rot-free computation for $\boldsymbol{h}_5$ at online phase.

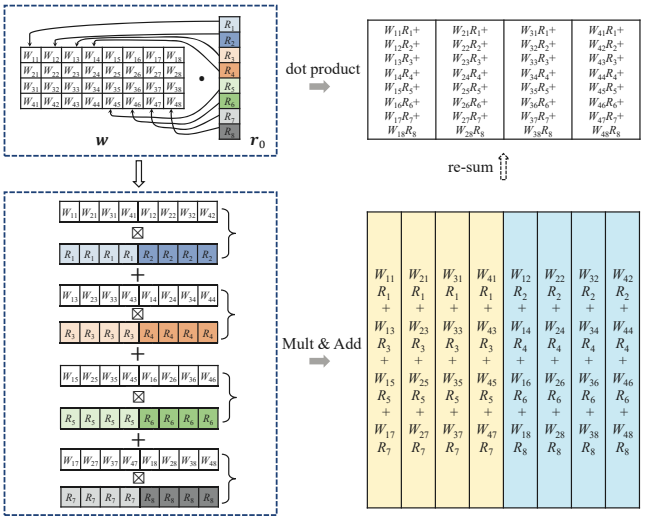

Figure 7. Relationship between FC and HE operations.

$[\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}$ and $[\boldsymbol{h}_3]_{\mathcal{S}}$ contain ciphertext with that amount. Then $[\boldsymbol{h}_5]_{\mathcal{S}}$ is sent to $\mathcal{S}$ which performs the decryption to get $\boldsymbol{h}_5$ and computes $\boldsymbol{k} * \boldsymbol{h}_5$ in plaintext. It is worth pointing out that $\mathcal{S}$ obtains $\boldsymbol{h}_5$ through a unidirectional online transmission from $\mathcal{C}$, which not only eliminates the multiplexing for getting $f'_{\mathrm{r}}(\boldsymbol{x}) \boxtimes \boldsymbol{x}$ but also avoids HE rotations with HE additions and multiplications both in $O(C_i/C_n)$. This makes the crypto complexity independent of kernel size (i.e., f_h and C_o), which contributes to significant cost reduction for the linear function $f_{\mathrm{c}}(\boldsymbol{y})$. The protocol for our Rot-free computation for $\boldsymbol{h}_5$ at online phase is described in Figure 6.

Putting things together. Let's put all pieces together to streamline FIT's joint optimization for $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ according to Eq. 5. Specifically, our protocol is divided into two phases namely offline computation and online computation. The offline computation involves an input-independent process that shares the $\boldsymbol{k} * \boldsymbol{r}_0$ and transmits prepared ciphertext namely $[\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}$ and $[\boldsymbol{h}_3]_{\mathcal{S}}$ from $\mathcal{S}$ to $\mathcal{C}$. Meanwhile, $\boldsymbol{k} * \boldsymbol{h}_4$ is locally computed by the server at offline phase. At online phase, the client and the server first engage in the OT-based module for getting $f'_{\mathrm{r}}(\boldsymbol{x})$ where $\mathcal{C}$ obtains its Boolean share $\boldsymbol{g}_0(\boldsymbol{x}) \in \mathbb{Z}_2^{C_i \times H_i \times W_i}$. Then the client computes $[\boldsymbol{h}_5]_{\mathcal{S}}$ according to Eq. 6, based on the ciphertext that are received at offline phase. Next $\mathcal{C}$ sends $[\boldsymbol{h}_5]_{\mathcal{S}}$ to $\mathcal{S}$ which preforms HE decryption and computes $\boldsymbol{k} * \boldsymbol{h}_5$. After that, the server shares $\boldsymbol{k} * \boldsymbol{h}_5$ with the client by sampling $\boldsymbol{shr}_{11}^{\mathsf{on}} \xleftarrow{\$} \mathbb{Z}_p^{C_o \times H_o \times W_o}$ and sending $\boldsymbol{shr}_{01}^{\mathsf{on}} = (\boldsymbol{k} * \boldsymbol{h}_5 - \boldsymbol{shr}_{11}^{\mathsf{on}})$ to the client. Finally, $\mathcal{S}$ sets its share of $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ as

$$\boldsymbol{shr}_1 \in \mathbb{Z}_p^{C_o \times H_o \times W_o} = \boldsymbol{shr}_{11}^{\mathsf{off}} + \boldsymbol{shr}_{12}^{\mathsf{off}} + \boldsymbol{shr}_{11}^{\mathsf{on}}$$

where $\boldsymbol{shr}_{11}^{\mathsf{off}} \in \mathbb{Z}_p^{C_o \times H_o \times W_o} = \boldsymbol{k} * \boldsymbol{h}_4$, while $\mathcal{C}$ sets its share of $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ as

$$\boldsymbol{shr}_0 \in \mathbb{Z}_p^{C_o \times H_o \times W_o} = \boldsymbol{shr}_{02}^{\mathsf{off}} + \boldsymbol{shr}_{01}^{\mathsf{on}}$$

and it is obvious to find that

$$\begin{aligned} &\boldsymbol{shr}_1 + \boldsymbol{shr}_0 \\ &= \boldsymbol{shr}_{11}^{\mathsf{off}} + (\boldsymbol{shr}_{12}^{\mathsf{off}} + \boldsymbol{shr}_{02}^{\mathsf{off}}) + (\boldsymbol{shr}_{11}^{\mathsf{on}} + \boldsymbol{shr}_{01}^{\mathsf{on}}) \\ &= \boldsymbol{k} * \boldsymbol{h}_4 + \boldsymbol{k} * \boldsymbol{r}_0 + \boldsymbol{k} * \boldsymbol{h}_5 = f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x})) \mod p \end{aligned}$$

The reason of sharing $\boldsymbol{k} * \boldsymbol{h}_5$ is to make $\boldsymbol{shr}_1$ hold the pregeneration property as $\boldsymbol{x}_1$, which enables subsequent calling of our joint optimization module for computing another $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$.

Additionally, the composite function of fully connection and ReLU namely $f_{\mathrm{w}}(f_{\mathrm{r}}(\boldsymbol{x}))$, with weight matrix $\boldsymbol{w}$, is computed similarly to $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ by replacing $\boldsymbol{k}$ and convolution " $*$ " with $\boldsymbol{w}$ and dot product " $\cdot$ ", respectively. Specifically, the offline sharing of $\boldsymbol{w} \cdot \boldsymbol{r}_0$, in contrast to the offline sharing of $\boldsymbol{k} * \boldsymbol{r}_0$ in $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$, utilizes the column accumulation of $\boldsymbol{w}$ for computing dot product, as shown in Figure 7. Therefore, the client first packs $\lfloor N/n_o \rfloor$ length-n_o vectors, each of which contains copies of one element in $\boldsymbol{r}_0 \in \mathbb{Z}_p^{n_i}$, and encrypts them into one ciphertext. The total $n_i n_o / N$ ciphertext are then sent to $\mathcal{S}$. The server performs HE multiplications and additions both in $O(n_i n_o)$, and produces a ciphertext that contains partial sums of $\boldsymbol{w} \cdot \boldsymbol{r}_0$. That ciphertext is masked and sent to the client, which conducts one decryption and plaintext re-summing to get its share of $\boldsymbol{w} \cdot \boldsymbol{r}_0$. In comparison, state-of-the-art frameworks usually let the client encrypt $\boldsymbol{r}_0$ and the fully connection is then performed at $\mathcal{S}$ through massive amount of HE operations with rotations in $O(n_i n_o - \log n_o)$ and multiplications in $O(n_i n_o)$ [13], or trading rotations by HE extractions and decryptions both in $O(n_o)$ with multiplications in $O(n_i n_o)$ [27].

Complexity analysis. We now give concrete complexities of FIT's computation for $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$. In the offline phase, $[\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}$ and $[\boldsymbol{h}_3]_{\mathcal{S}}$, each includes $\lceil C_i H_i W_i / N \rceil$ ciphertext, are sent from $\mathcal{S}$ to $\mathcal{C}$. Meanwhile, to share $\boldsymbol{k} * \boldsymbol{r}_0$, the client transforms $\boldsymbol{r}_0$ into $\widehat{\boldsymbol{r}_0}$ in accordance with Figure 4

TABLE 2. COMPARISON OF COMPUTATION COMPLEXITY TO OBTAIN SHARES OF $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$. THE COMMON PART TO GET SHARES OF DERIVATIVE OF RELU IS EXCLUDED.

Frameworks	Offline Computation				Online Computation					
	#Enc	#Mult	#Dec	#Add	Mux	#Rot	#Enc	#Mult	#Dec	#Add
CrypTFlow2	-	-	-	-	✓	$\geq \frac{(f_h^2-1)C_i}{C_n} + C_o - \frac{C_o}{C_n}$	$\frac{C_i}{C_n}$	$\frac{f_h^2 C_i C_o}{C_n}$	$\frac{C_o}{C_n}$	$\frac{C_i + C_o C_i f_h^2}{C_n}$
FIT	$\frac{f_h^2 C_i}{C_n}$	$\frac{f_h^2 C_i C_o}{C_n}$	C_o	$\frac{f_h^2 C_i C_o}{C_n}$	✗	-	-	$2\frac{C_i}{C_n}$	$\frac{C_i}{C_n}$	$2\frac{C_i}{C_n}$

and encrypts $\widehat{\boldsymbol{r}_0}$ into $[\widehat{\boldsymbol{r}_0}]_{\mathcal{C}}$ with $\lceil (f_h)^2 C_i / \lfloor N/H_o W_o \rfloor \rceil$ ciphertext. The $[\widehat{\boldsymbol{r}_0}]_{\mathcal{C}}$ is sent to the server, which conducts $\lceil (f_h)^2 C_i / \lfloor N/H_o W_o \rfloor \rceil$ HE multiplications and additions to form one shared ciphertext for each of C_o filters. Those C_o ciphertext are transmitted from $\mathcal{S}$ to $\mathcal{C}$, which does C_o decryptions and obtains $\boldsymbol{shr}_{02}^{\mathsf{off}}$. In the online phase, $\mathcal{C}$ and $\mathcal{S}$ first involve in the OT-based module for $f'_{\mathsf{r}}(\boldsymbol{x})$ which outputs $\boldsymbol{g}_0(\boldsymbol{x})$ for $\mathcal{C}$. After that, the client performs $2\lceil C_i H_i W_i / N \rceil$ HE multiplications and $2\lceil C_i H_i W_i / N \rceil$ additions to form $[\boldsymbol{h}_5]_{\mathcal{S}}$ with $\lceil C_i H_i W_i / N \rceil$ ciphertext. $[\boldsymbol{h}_5]_{\mathcal{S}}$ is then sent to $\mathcal{S}$. The server does $\lceil C_i H_i W_i / N \rceil$ decryptions and obtains $\boldsymbol{k} * \boldsymbol{h}_5$, which is then shared with $\mathcal{C}$ and the client finally gets its share of $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ as $\boldsymbol{shr}_0$ while $\mathcal{S}$ forms $\boldsymbol{shr}_1$.

As for the total computation complexity with both offline and online considered, we compare FIT with state-of-the-art CrypTFlow2 framework [17] as shown in Table 2. FIT demonstrates comparable or even lower complexity according to the following five observations. First, the number of HE Mult of FIT is comparable with that of [17]. Second, the number of HE Add of FIT is comparable with that of [17]. Third, the number of HE Dec of FIT at online computation is comparable with that of [17]. Fourth, by offsetting the HE Enc between FIT and [17], the remaining complexity of FIT namely $\frac{(f_h^2-1)C_i}{C_n}$ HE Enc plus C_o HE Dec is comparable or even lower than that of [17], namely the complexity of HE Rot, as the HE Rot is much expensive than HE Enc and HE Dec [13]. Finally, FIT also escapes the execution of OT-based Multiplexing (Mux), which is indispensable in [17] and thus brings another efficiency opportunity. Overall, FIT features with a much efficient online computation while makes the overall complexity comparable or even lower. Such property is reasonable and more preferred because in real-word MLaaS applications, the user always needs timely online response and the offline cost is more tolerable.

Furthermore, the complexities for computing $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$ is similarly analyzed as follows. As for the offline sharing of $\boldsymbol{w} \cdot \boldsymbol{r}_0$, $\mathcal{C}$ encrypts $\lceil n_i / \lfloor N/n_o \rfloor \rceil$ ciphertext with respect to $\boldsymbol{r}_0$ and sent them to $\mathcal{S}$. The server performs $\lceil n_i / \lfloor N/n_o \rfloor \rceil$ HE multiplications and $\lceil n_i / \lfloor N/n_o \rfloor \rceil$ to produces a masked ciphertext that contains partial sums of $\boldsymbol{w} \cdot \boldsymbol{r}_0$. That ciphertext is then sent to the client, which conducts one decryption and plaintext re-summing to get its share of $\boldsymbol{w} \cdot \boldsymbol{r}_0$. As for the online computation, after the OT-based module for getting share of $f'_{\mathsf{r}}(\boldsymbol{x})$ namely $\boldsymbol{g}_0(\boldsymbol{x})$, the client performs $\lceil n_i / N \rceil$ HE multiplications and $\lceil n_i / N \rceil$ additions to form $[\boldsymbol{h}_5]_{\mathcal{S}}$ with $\lceil n_i / N \rceil$ ciphertext. $[\boldsymbol{h}_5]_{\mathcal{S}}$ is then sent to $\mathcal{S}$. The server does $\lceil n_i / N \rceil$ decryptions and obtains $\boldsymbol{w} \cdot \boldsymbol{h}_5$, which is then shared with $\mathcal{C}$ and the client finally gets its share of $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$ as $\boldsymbol{shr}_0$ while $\mathcal{S}$ forms $\boldsymbol{shr}_1$.

3.3. Model Adjustments for Best Utilization

A neural network always contains other functions than $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ and $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$ (e.g., maxpooling, meanpooling, and batch normalization). Therefore, we propose to do the network adjustments to maximize the utilization of FIT's joint optimization for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ and $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$. The basic idea is to equivalently reassemble functions in the neural network such that $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ and $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$ appear as much as possible. To this end, we identify three function blocks where ReLU and Conv are usually separated by other functions, and we aim to equalize such function blocks to make ReLU and Conv adjacent. First, the maxpooling (MaxPool), $f_{\mathsf{mx}}(\boldsymbol{x})$, usually appears between ReLU and Conv as ReLU→MaxPool→Conv which is indeed equavalent with MaxPool→ReLU→Conv since $f_{\mathsf{r}}(f_{\mathsf{mx}}(\boldsymbol{x})) = f_{\mathsf{mx}}(f_{\mathsf{r}}(\boldsymbol{x}))$. While this ReLU-MaxPool conversion has been applied in optimizations for GC-based and SS-based nonlinear functions [39], [40], as well as in plaintext computation for neural networks [41], they only consider the sequential computation for stacked functions in a neural network, and we further gain efficiency boost on top of such conversion via FIT's joint optimization for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$. Second, the meanpooling (MeanPool), $f_{\mathsf{mn}}(\boldsymbol{x})$, usually appears between ReLU and Conv as ReLU→MeanPool→Conv, and we separately perform MeanPool over $\boldsymbol{h}_5$, $\boldsymbol{h}_4$ and $\boldsymbol{r}_0$ based on Eq. (5) in the joint optimization process since

$$\begin{aligned} f_{\mathsf{c}}(f_{\mathsf{mn}}(f_{\mathsf{r}}(\boldsymbol{x}))) &= \boldsymbol{k} * f_{\mathsf{mn}}(\boldsymbol{h}_5 + \boldsymbol{h}_4 + \boldsymbol{r}_0) \\ &= \boldsymbol{k} * f_{\mathsf{mn}}(\boldsymbol{h}_5) + \boldsymbol{k} * f_{\mathsf{mn}}(\boldsymbol{h}_4) + \boldsymbol{k} * f_{\mathsf{mn}}(\boldsymbol{r}_0) \end{aligned}$$

Specifically, the offline sharing for $\boldsymbol{k} * \boldsymbol{r}_0$ is replaced by that for $\boldsymbol{k} * f_{\mathsf{mn}}(\boldsymbol{r}_0)$ where the client first conducts MeanPool over $\boldsymbol{r}_0$ to produce a new $\boldsymbol{r}_0$ with smaller dimensions, and that new $\boldsymbol{r}_0$ then acts as the $\boldsymbol{r}_0$ in Figure 5 to complete the sharing process. Meanwhile, the server computes $\boldsymbol{k} * f_{\mathsf{mn}}(\boldsymbol{h}_4)$ instead of $\boldsymbol{k} * \boldsymbol{h}_4$. In the online phase, the server calculates $\boldsymbol{k} * f_{\mathsf{mn}}(\boldsymbol{h}_5)$ instead of $\boldsymbol{k} * \boldsymbol{h}_5$ after it obtains $\boldsymbol{h}_5$. Third, the batch normalization, $f_{\mathsf{bn}}(\boldsymbol{x})$, usually appears after Conv as and is specified by a constant tuple $(\boldsymbol{\mu}, \boldsymbol{\theta}) \in \mathbb{Z}_p^{C_o} \times \mathbb{Z}_p^{C_o}$ which scales and shifts each of C_o 2-dimension convolution output by one element in $\boldsymbol{\mu}$ and $\boldsymbol{\theta}$, respectively [27]. Since $f_{\mathsf{bn}}(\boldsymbol{x})$ only involves scalar multiplication and addition, it is easy to combine batch normalization (BN) with Conv in FIT's offline phase where each of C_o filters in $\boldsymbol{k}$ is multiplied with one element in $\boldsymbol{\mu}$, and each of C_o 2-dimension convolution output in $\boldsymbol{k} * \boldsymbol{h}_4$ is added with one element in $\boldsymbol{\theta}$. In this way, the Conv→BN is able to integrate with ReLU based on our joint optimization for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ (or $f_{\mathsf{w}}(f_{\mathsf{r}}(\boldsymbol{x}))$).

Additionally, once $\mathcal{C}$ launches a query to get the output of $\mathcal{S}$'s neural model, the private input $\boldsymbol{x}$ is firstly fed to Conv rather than ReLU, which makes us unable to directly utilize optimization for $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$. We address the Conv with $\mathcal{C}$'s input by computing $\boldsymbol{k} * \{(\boldsymbol{x} - \boldsymbol{r}_0) + \boldsymbol{r}_0\}$ where $\boldsymbol{k} * \boldsymbol{r}_0$ is shared in a way similar with that in Figure 5, and $\boldsymbol{x} - \boldsymbol{r}_0$ is sent, at online phase, from the client to the server which calculates and shares $\boldsymbol{k} * (\boldsymbol{x} - \boldsymbol{r}_0)$ in a way similar with $\mathcal{S}$'s computation and sharing of $\boldsymbol{k} * \boldsymbol{h}_5$ in $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$.

Last but not least, our joint optimization for composite function $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ assumes the pregeneration property of $\mathcal{S}$'s shares with respect to input namely $\boldsymbol{x}_1$ and $\boldsymbol{g}_1(\boldsymbol{x})$. We deal with the non-pregeneration case, where the shares of neither party towards $\boldsymbol{x}$ are input-independent, by running $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$ with an offline-included online phase and a final sharing of $\boldsymbol{k} * \boldsymbol{h}_5$. In this way, $\mathcal{S}$ is able to form pregenerated shares for $f_{\mathrm{c}}(f_{\mathrm{r}}(\boldsymbol{x}))$, which are eligible to feed in our joint optimization for computing subsequent $f_{\mathrm{c}}(f_{\mathrm{r}}(\cdot))$.

3.4. Security Analysis

We aim to prove the security of Rot-free sharing for $\boldsymbol{k} * \boldsymbol{r}_0$ at offline phase as shown in Figure 5 and Rot-free computation for $\boldsymbol{h}_5$ at online phase as shown in Figure 6. Since other functions are directly implemented based on 2PC protocols from [17] and the output of our offline and online computation is either randomly shared or to be randomly shared, the security of computing entire model is guaranteed after composition because a protocol that ends with secure re-sharing of output is universally composable [12], [42].

THEOREM 1. *The protocol in Figure 5 is secure in the presence of semi-honest adversaries, if E(·) is semantically secure.*

$Proof$. Our security proof is based on the ideal/real-world paradigm [37] discussed in Section 2.2: in the real world, $\mathcal{S}$ and $\mathcal{C}$ interact with each other according to protocol specification, while in the ideal world, parties have access to a trusted third party TTP that implements $\mathcal{F}$ which returns a random share $\boldsymbol{shr}_{12}^{\mathsf{off}} \in \mathbb{Z}_p^{C_o \times H_o \times W_o}$ to $\mathcal{S}$ and a share $\boldsymbol{shr}_{02}^{\mathsf{off}} \in \mathbb{Z}_p^{C_o \times H_o \times W_o}$ satisfying $\boldsymbol{shr}_{12}^{\mathsf{off}} + \boldsymbol{shr}_{02}^{\mathsf{off}} = \boldsymbol{k} * \boldsymbol{r}_0 \bmod p$, given $\boldsymbol{k} \in \mathbb{Z}_p^{C_o \times C_i \times H_i \times W_i}$ from the server and $\boldsymbol{r}_0 \in \mathbb{Z}_p^{C_i \times H_i \times W_i}$ from the client. The execution in both worlds are coordinated by environment $\mathcal{E}$ which selects inputs to parties and acts as a distinguisher between real and ideal executions. Our goal is to show that the adversary's view in real world is indistinguishable to that in ideal world.

Security against a semi-honest server. We prove security against a semi-honest server by constructing an ideal-world simulator Sim that performs as follows:

(1) receives $\boldsymbol{k}$ from $\mathcal{E}$, Sim sends $\boldsymbol{k}$ to TTP;
(2) starts running $\mathcal{S}$ on input $\boldsymbol{k}$;
(3) constructs $[\widehat{\boldsymbol{r}_0}']_{\mathcal{S}im} \leftarrow E(pk_{Sim}, \{\mathbf{0}\})$ where pk_{Sim} is randomly generated by Sim;
(4) sends $[\widehat{\boldsymbol{r}_0}']_{\mathcal{S}im}$ to $\mathcal{S}$;
(5) outputs whatever $\mathcal{S}$ outputs.

$\mathcal{S}$'s view in real execution is $E(pk_C, \widehat{\boldsymbol{r}_0})$, which is computationally indistinguishable from its view in ideal execution, $E(pk_{Sim}, \{\mathbf{0}\})$, based on the semantic security of $E(\cdot)$. Therefore the output distribution of $\mathcal{E}$ in real world is computationally indistinguishable from that in ideal world.

Security against a semi-honest client. Next, we prove security against a semi-honest client by constructing an ideal-world simulator Sim that works as follows:

(1) receives $\boldsymbol{r}_0$ from environment $\mathcal{E}$, Sim sends $\boldsymbol{r}_0$ to TTP and gets the result $\boldsymbol{shr}_{02}^{\mathsf{off}}$;
(2) starts running $\mathcal{C}$ on input $\boldsymbol{r}_0$, and receives $[\widehat{\boldsymbol{r}_0}]_{\mathcal{C}}$;
(3) randomly splits $\boldsymbol{shr}_{02}^{\mathsf{off}}$ into $\boldsymbol{shr}'^{\mathsf{off}}_{02}$ subjected to $\boldsymbol{shr}_{02}^{\mathsf{off}} = \sum \boldsymbol{shr}'^{\mathsf{off}}_{02}$;
(4) encrypts $\boldsymbol{shr}'^{\mathsf{off}}_{02}$ using $\mathcal{C}$'s public key and returns $[\boldsymbol{shr}'^{\mathsf{off}}_{02}]_{\mathcal{C}}$ to $\mathcal{C}$;
(5) outputs whatever $\mathcal{C}$ outputs.

$\mathcal{C}$'s view in real execution is $\sum(\widehat{\boldsymbol{r}_0} \boxtimes \widehat{\boldsymbol{k}}) - \widetilde{\boldsymbol{shr}}_{12}^{\mathsf{off}}$ while its view in ideal execution is $\boldsymbol{shr}'^{\mathsf{off}}_{02}$. Thus we only need to show that any element in $\sum(\widehat{\boldsymbol{r}_0} \boxtimes \widehat{\boldsymbol{k}}) - \widetilde{\boldsymbol{shr}}_{12}^{\mathsf{off}}$ is indistinguishable from a random number in $\boldsymbol{shr}'^{\mathsf{off}}_{02}$. This is clearly true since $\widetilde{\boldsymbol{shr}}_{12}^{\mathsf{off}}$ is randomly chosen. At the end of simulation, $\mathcal{C}$ outputs $\boldsymbol{shr}_{02}^{\mathsf{off}} = \sum \boldsymbol{shr}'^{\mathsf{off}}_{02}$, which is the same as real execution. Therefore we claim that the output distribution of $\mathcal{E}$ in real world is computationally indistinguishable from that in ideal world.

THEOREM 2. *The protocol in Figure 6 is secure in the presence of semi-honest adversaries, if OT and E(·) are semantically secure.*

$Proof$. Similar with THEOREM 1, our security proof is based on the ideal/real-world paradigm: in real world, $\mathcal{S}$ and $\mathcal{C}$ interact with each other according to protocol specification, while in ideal world, parties have access to a trusted third party TTP that implements $\mathcal{F}$ which returns $\boldsymbol{h}_5$ to $\mathcal{S}$, given the inputs from $\mathcal{S}$ and $\mathcal{C}$ as specified in Figure 6. Our goal is to show that the adversary's view in real world is indistinguishable to that in ideal world.

Security against a semi-honest server. We prove security against a semi-honest server by constructing an ideal-world simulator Sim that performs as follows:

(1) receives $\boldsymbol{x}_1$ and $\boldsymbol{g}_1(\boldsymbol{x})$ from environment $\mathcal{E}$, Sim sends $\boldsymbol{x}_1$ and $\boldsymbol{g}_1(\boldsymbol{x})$ to TTP and gets the result $\boldsymbol{h}_5$;
(2) starts running $\mathcal{S}$ with $\boldsymbol{x}_1$ and $\boldsymbol{g}_1(\boldsymbol{x})$, and receives $\boldsymbol{g}_0(\boldsymbol{x})$;
(3) encrypts $\boldsymbol{h}_5$ using $\mathcal{S}$'s public key and returns $[\boldsymbol{h}_5]_{\mathcal{S}}$ to $\mathcal{S}$;
(4) outputs whatever $\mathcal{S}$ outputs.

On the one hand, the semantic security of OT guarantees the computational indistinguishability of messages to $\mathcal{S}$ for computing $f'_{\mathrm{r}}(\boldsymbol{x})$ in real and ideal executions. On the other hand, $\mathcal{S}$ outputs $\boldsymbol{h}_5$ at the end of simulation, which is the same as real execution. Therefore we claim that the output distribution of $\mathcal{E}$ in real world is computationally indistinguishable from that in ideal world.

Security against a semi-honest client. We prove security against a semi-honest client by constructing an ideal-world simulator Sim that performs as follows:

(1) receives $\boldsymbol{x}_0$, $\boldsymbol{r}_0$, $[\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}$, and $[\boldsymbol{h}_3]_{\mathcal{S}}$ from environment $\mathcal{E}$, Sim sends them to TTP;

(2) starts running $\mathcal{C}$ on input $\boldsymbol{x}_0$, $\boldsymbol{r}_0$, $[\boldsymbol{g}_1(\boldsymbol{x})]_{\mathcal{S}}$, and $[\boldsymbol{h}_3]_{\mathcal{S}}$;
(3) outputs whatever $\mathcal{C}$ outputs.

$\mathcal{C}$'s view in real execution includes messages for computing $f'_{\mathsf{r}}(\boldsymbol{x})$, which is computationally indistinguishable from its view in ideal execution based on the semantic security of OT. Therefore the output distribution of $\mathcal{E}$ in real world is computationally indistinguishable from that in ideal world.

4. Evaluation

We present in this section the performance results of FIT. Specifically, we implement FIT based on the open-sourced code from CrypTFlow2 [17] and all experiments are run in a LAN setting with gigabit bandwidth. Each machine possesses a CPU with Intel Core i7-11370H@3.3GHz, and the system memory is 38.9GB. Meanwhile, we evaluate performance of making inference for MNIST-scale [43] data over model from [44] which we denote as M1, and CIFAR10-scale [45] and ImageNet-scale [46] data over VGG-16 [3], VGG-19 [3], ResNet-18 [4], ResNet-34 [4], DenseNet-161 [47], and DenseNet-169 [47] architectures. The concrete model configurations are described in Appendix A.

Concretely, Section 4.1 illustrates the online performance of FIT compared to state-of-the-art works, including the cost of our composite module for computing $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ and the efficiency over modern networks that integrate computation module of FIT. Section 4.2 breaks down the cost over modern networks, including the offline overhead, and discusses in detail the impact of FIT on various function dimensions, which help to understand FIT's adaptability when applied to different neural models.

4.1. Online Performance

Recall that the proposed FIT features with light-weight running-time complexity to compute composite function $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$, which is independent of filter size namely f_h and C_o. As such we demonstrate in this section the concrete performance, in the online phase, over individual function blocks as well as complete neural models when plugging in FIT's composite modules. Specifically, the goal is to output shares of $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ given the shares of $\boldsymbol{x}$ with size of $H_i \times W_i @ C_i$. We measure the total communication including all the messages sent by $\mathcal{C}$ and $\mathcal{S}$ and the end-to-end running time including the time of transferring messages through the LAN.

Microbenchmarks. Table 3 demonstrates the performance of FIT's composite module compared to the state-of-the-art approaches, CrypTFlow2 [17], Cheetah [27], and GAZELLE [13]. The tested dimensions are picked from modern networks such as ResNet and DenseNet. Specifically, with the filter-independent crypto computation, FIT shows up to 29.9× speedup, 3.7× speedup, and 300× speedup on various function sizes, compared to CrypTFlow2, Cheetah, and GAZELLE, respectively. Meanwhile, FIT removes two simultaneous communication rounds for multiplexing, and reduces transmission load in terms of OT for multiplexing and ciphertext for sharing. All together, FIT computes composite functions in a much more efficient way, without balancing the communication cost both in round and transferred data.

Over modern networks. Table 4 shows the performance over different modern networks, with various datasets, by plugging in FIT's composite modules as well as applying the proposed network adaptations. The key takeaways are summarized as follows. First, as for M1 with MNIST, FIT demonstrates 35.5× speedup for computation and less cost for communication. Such performance gain is attributed to large kernel sizes (i.e., $f_h = 7$) and large output channels (e.g., $C_o = 144$ and $C_o = 192$) in M1, which makes the cost of CrypTFlow2 much more than that of FIT as indicated in Table 2. Second, as for VGG-16 and ResNet-18 with CIFAR10, FIT shows 4.5× and 8.9× computational speedup, respectively. As for the communication cost, FIT transfers more data than [17] especially over VGG-16. The main reason behind it is the inclusion of offline process due to the non-pregenerated property of shares of MaxPool functions after applying FIT's network adaptation, which introduces extra communication load. That offline inclusion has less impact of communication cost on ResNet-18 since there are less MaxPool functions. It implies that FIT is more suitable with less Maxpool functions in terms of large-size network with small-scale data.

Third, as for VGG-16 and ResNet-18 with ImageNet, FIT shows 8.4× and 5.5× computational speedup with reduced communication cost, respectively. The main reason is that the efficiency harvest of FIT's model adaptation towards Maxpool functions is more significant than the cost of corresponding offline inclusion. This property suggests that FIT is applicable to large-size network with large-scale data. Similar observations are found over VGG-19 and ResNet-34. Finally, FIT is integrated into much deeper networks DenseNet-161 and DenseNet-169 to evaluate its adaptability with more complex architectures. Specifically, FIT demonstrates 11× speedup and 18.2× speedup over DenseNet-161 and DenseNet-169, respectively. The BN→ReLU→Conv for each "conv" layer from DenseNet makes FIT's composite module for $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ naturally suitable to accelerate the privacy-preserving computation with almost no offline inclusion as needed in ResNet and VGG. Furthermore, the relatively smaller input channels and output channels in DenseNet-169 namely smaller C_i and C_o result in less computation cost of FIT compared with its overhead for DenseNet-161, which contributes to more speedup over DenseNet-169.

4.2. Offline Included Performance Breakdown

In order to concretely understand FIT's adaptability over different function dimensions, including the offline overhead, we further break down the performance of tested networks in function wise. As such, Figure 8 to Figure 10 demonstrate the accumulated time as well as accumulated communication cost both in online and offline phases. Here the value at each function is the cost of current function

TABLE 3. ONLINE COMPUTATION OF $f_{\mathsf{c}}(f_{\mathsf{r}}(\boldsymbol{x}))$ WITH VARIOUS SIZES. THE OT-BASED DERIVATIVE IS EXCLUDED FOR A FAIR COMPARISON.

$H_i \times W_i@C_i, s$ $f_h \times f_h@C_o$	Time (ms)				Communication (MB)		
	FIT	CrypTFlow2 [17]	Cheetah [27]	GAZELLE [13]	FIT	CrypTFlow2 [17]	Cheetah [27]
28×28@512, 1 1×1@128	207	4351 Speedup: 21×	643 Speedup: 3.1×	Speedup: 65×	19.57	29	10.47
28×28@512, 1 1×1@256	283	8478 Speedup: 29.9×	1060 Speedup: 3.7×	Speedup: 110×	20.39	32.59	14.8
14×14@1024, 1 1×1@256	246	4362 Speedup: 17.7×	513 Speedup: 2×	Speedup: 150×	9.8	14.91	10.81
14×14@1024, 1 1×1@512	423	8728 Speedup: 20.6×	1108 Speedup: 2.6×	Speedup: 300×	10.24	16.29	18.54
28×28@320, 1 1×1@128	152	2741 Speedup: 18×	383 Speedup: 2.5×	Speedup: 70×	12.44	19.68	8.14
28×28@480, 1 1×1@128	211	4164 Speedup: 19.7×	458 Speedup: 2.1×	Speedup: 68×	18.07	28.14	10.05

TABLE 4. ONLINE COMPUTATION OF VARIOUS NEURAL MODELS.

Dataset	Model	Time (s)		Communication (MB)	
		FIT	CrypTFlow2 [17]	FIT	CrypTFlow2 [17]
MNIST	M1	4.2 Speedup: 35.5×	149.49	323.11	347.2
CIFAR10	VGG-16	10.87 Speedup: 4.5×	49.83	895.81	337.23
	VGG-19	12.56 Speedup: 6.4×	80.85	920.06	367.74
	ResNet-18	7.931 Speedup: 8.9×	70.89	188.96	157.33
	ResNet-34	16.49 Speedup: 7×	116.61	271.03	275.8
	DenseNet-161	41.96 Speedup: 11×	462.78	1762.3	1947.7
	DenseNet-169	26.83 Speedup: 18.2×	489.5	1171.23	1285.98
ImageNet	VGG-16	213 Speedup: 8.4×	1790.181	15161.82	16311.65
	VGG-19	235.84 Speedup: 12×	2848.62	15910.08	17748.17
	ResNet-18	51.6 Speedup: 5.5×	284.095	3698.71	3944.66
	ResNet-34	77.46 Speedup: 8.6×	670.6	5331.75	5165.5

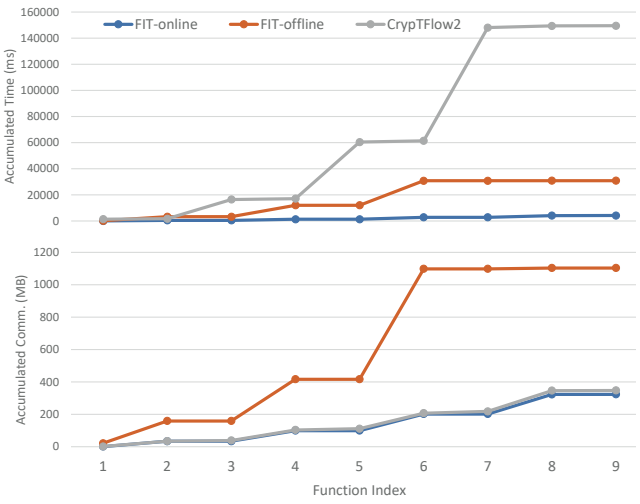

Figure 8. Performance breakdown of M1 with MNIST.

plus that of previous function. The variation of such accumulation helps to identify how the cost of each function contributes to the global overhead. Specifically, as for the M1 with MNIST in Figure 8, FIT's cost mainly comes from its offline communication for the second function, which takes 38.6MB. This is because the offline communication is proportional to the number of output channels C_o which is 100 in our case, which is in sharp contrast to that of the first function namely one. This results in the sudden increase of communication for the second function.

As for the VGG-16 with CIFAR10 in Figure 9(a), the communication jump at online phase happens in fourth, ninth, 16-th, 23-th and 30-th functions, which adopts offline inclusion. The offline inclusion needs to incorporate the whole offline cost in the online phase, which introduces noticeable communication overhead. As for the communication cost in offline phase, the jump happens in functions which involve the offline process. Meanwhile, as number of output channel is large, such as 128, 256 and 512, these jumps are non-negligible. As for the ResNet-18 with CIFAR10 in Figure 9(b), we pay attention to FIT's offline communication cost, which continuously increases as network goes deeper. The reasons lie in two folds. First, ResNet-18 mainly includes a stack of $f_{\mathsf{c}}(f_{\mathsf{r}}(\cdot))$ namely convolution layer, with few MaxPool functions. This structure enables a wide application of our proposed module for $f_{\mathsf{c}}(f_{\mathsf{r}}(\cdot))$. Since the offline cost is proportional to the number of output

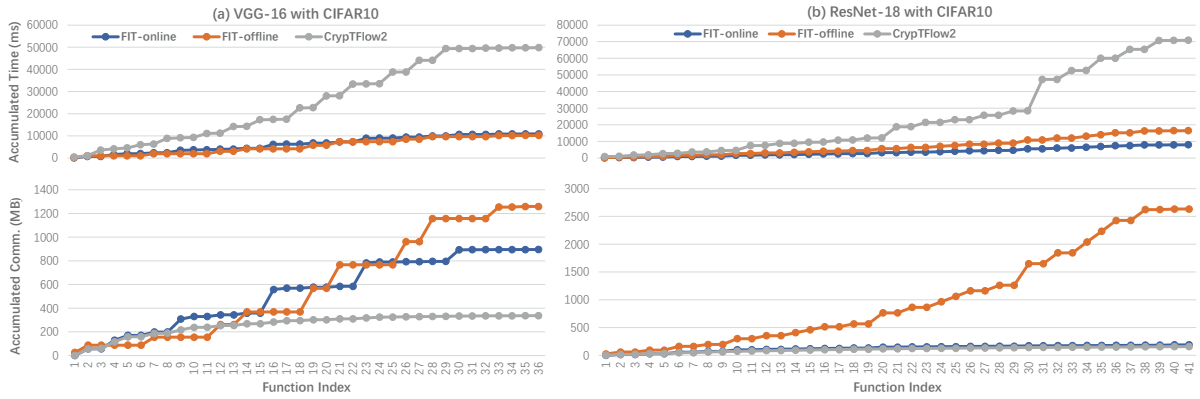

Figure 9. Performance breakdown of VGG-16 and ResNet-18 with CIFAR10.

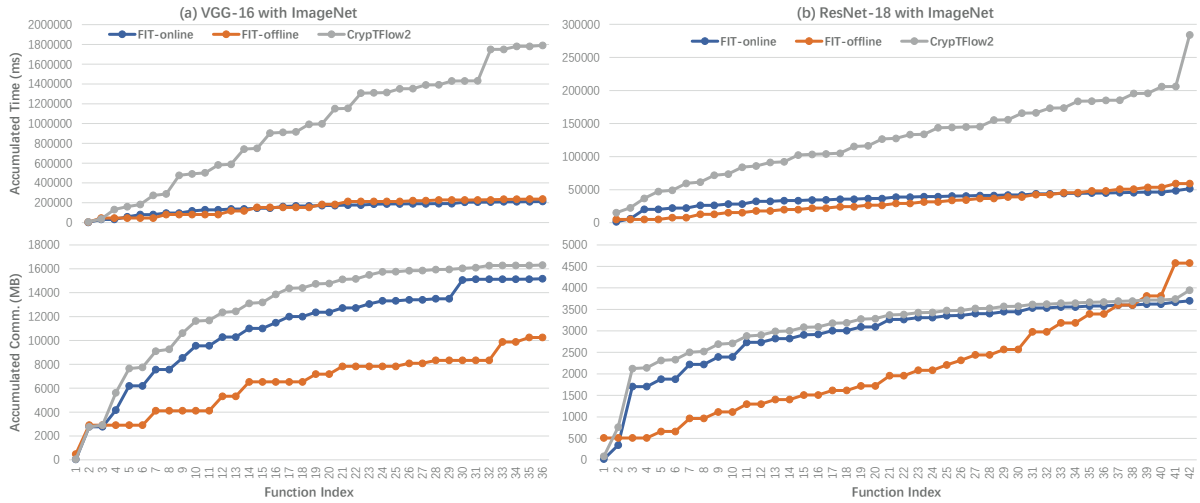

Figure 10. Performance breakdown of VGG-16 and ResNet-18 with ImageNet.

channels which is always several hundreds in our case, the communication shows a persistent increase. Second, the increase for 20-th, 30-th, 32-th, 34-th, 35-th, 36-th, 38-th functions is more steep since these places either involve bypass connection which needs double computation of our composite module, or have maximum of output channel which results in maximum number of ciphertext to be transmitted.

Furthermore, we look at the performance breakdown of networks with ImageNet. As for the VGG-16 shown in Figure 10(a), the communication variation changes among FIT-online, FIT-offline, and [17], compared with that using CIFAR10. The main reason is due to increase of input scale from CIFAR10 to ImageNet, which makes our network adaptation for the five MaxPool functions, together with the composite computation, benefits a lot from smaller input size of subsequent ReLU, while [17] involves much larger data scale to get corresponding ReLU and Conv. As for the ResNet-18 in Figure 10(b), a similar communication variation is also observed while the reason is not due to network adaptation for MaxPool functions but due to strided convolution and the relatively pure stake of Conv and ReLU. First, the strided convolution leads to a series of decomposed convolution in [17], which involves a series calls of individual Conv computation. Although the data scale of such decomposed convolution is smaller compared to the original one, the large input size limits the computational benefit. In contrast, FIT is naturally suitable for strided convolution with less computation. Second, the large-size input itself obviously increases the cost for computing each function in [17], while FIT's cost, especially the communication overhead, is not highly sensitive with input scale, but with the constant number of output channels, which helps to mitigate the cost of in both online and offline phases. Similar observations are found over VGG-19 and ResNet-34 as shown in Figure 11 and Figure 12.

Finally, as for the performance breakdown over much deeper networks DenseNet-161 and DenseNet-169, as demonstrated in Figure 13 and Figure 14, FIT always keeps its performance advantage at online phase with respect to both computation and communication. While FIT needs more offline communication at bottleneck layers such as the communication cost for the 226-th function from DenseNet-161 and the one for the 210-th function from DenseNet-169

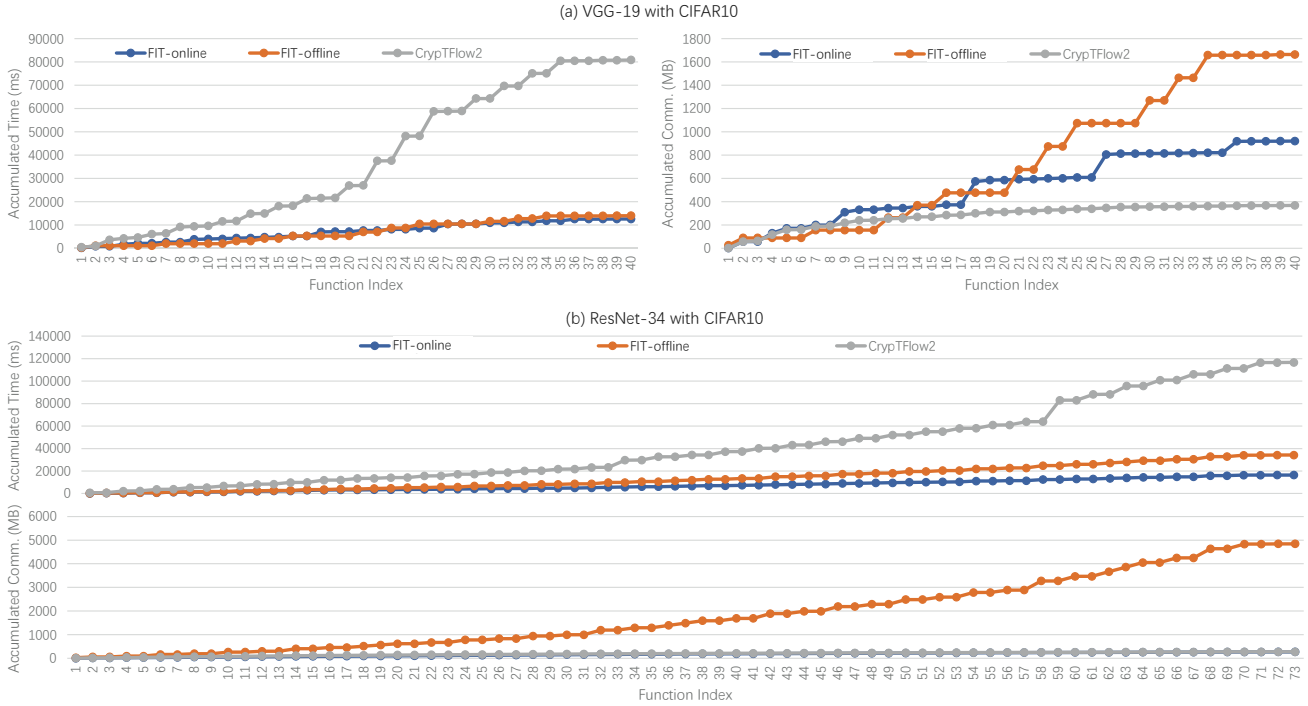

Figure 11. Performance Breakdown of VGG-19 and ResNet-34 with CIFAR10.

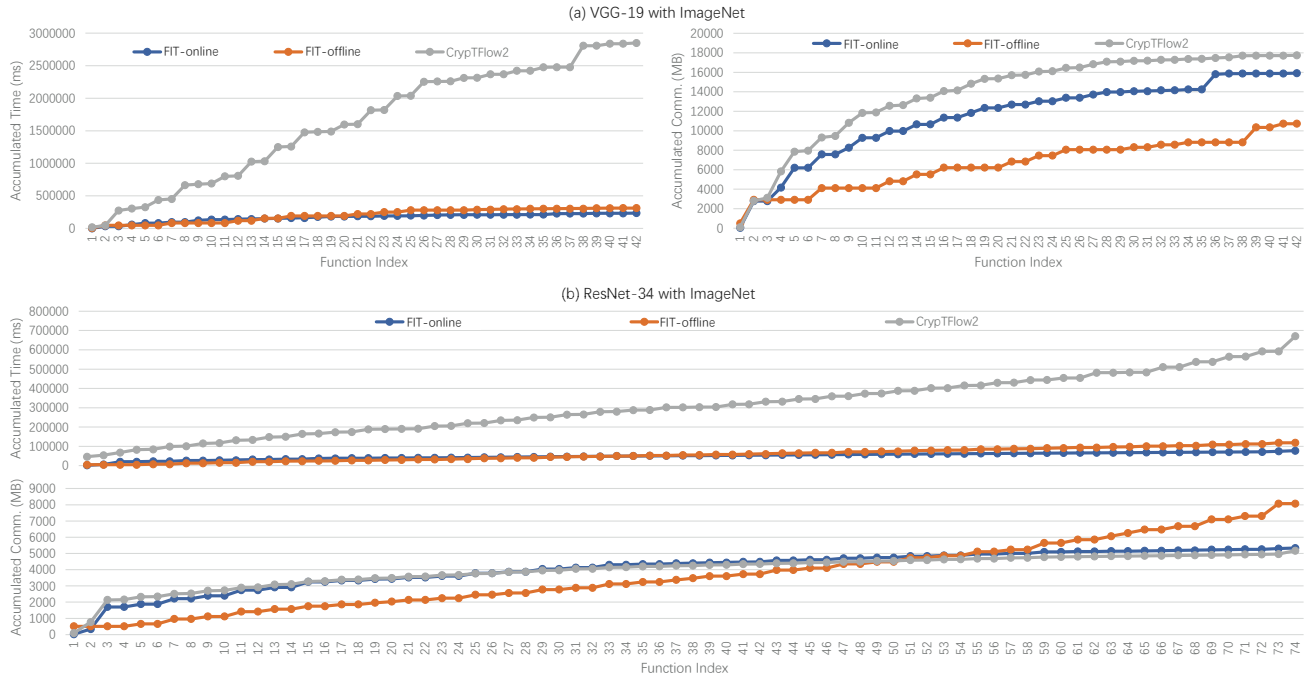

Figure 12. Performance Breakdown of VGG-19 and ResNet-34 with ImageNet.

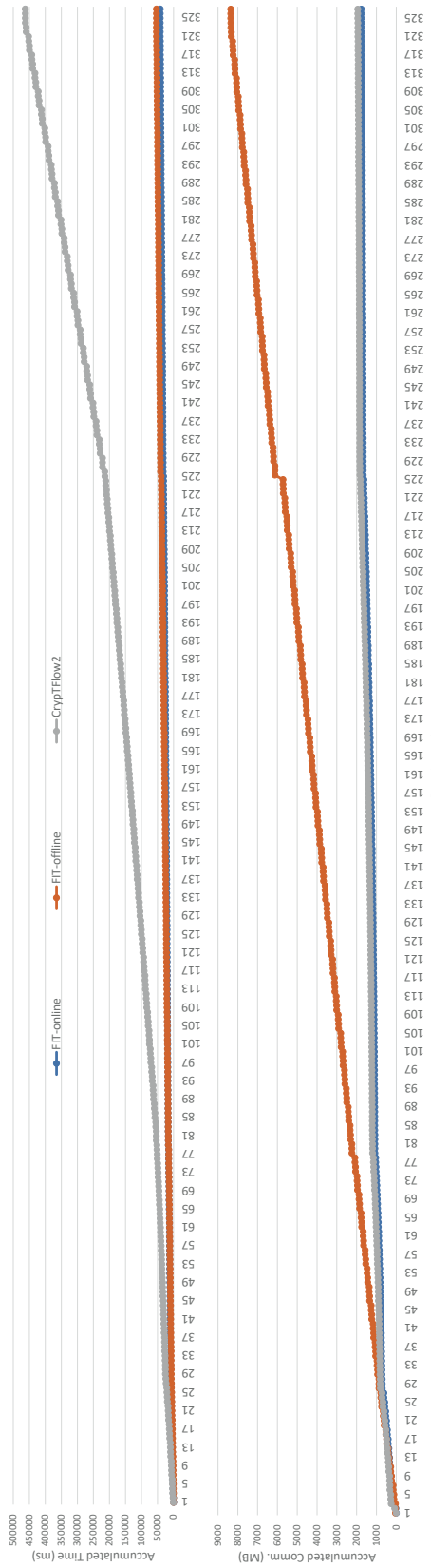

Figure 13. Performance Breakdown of DenseNet-161 with CIFAR10.

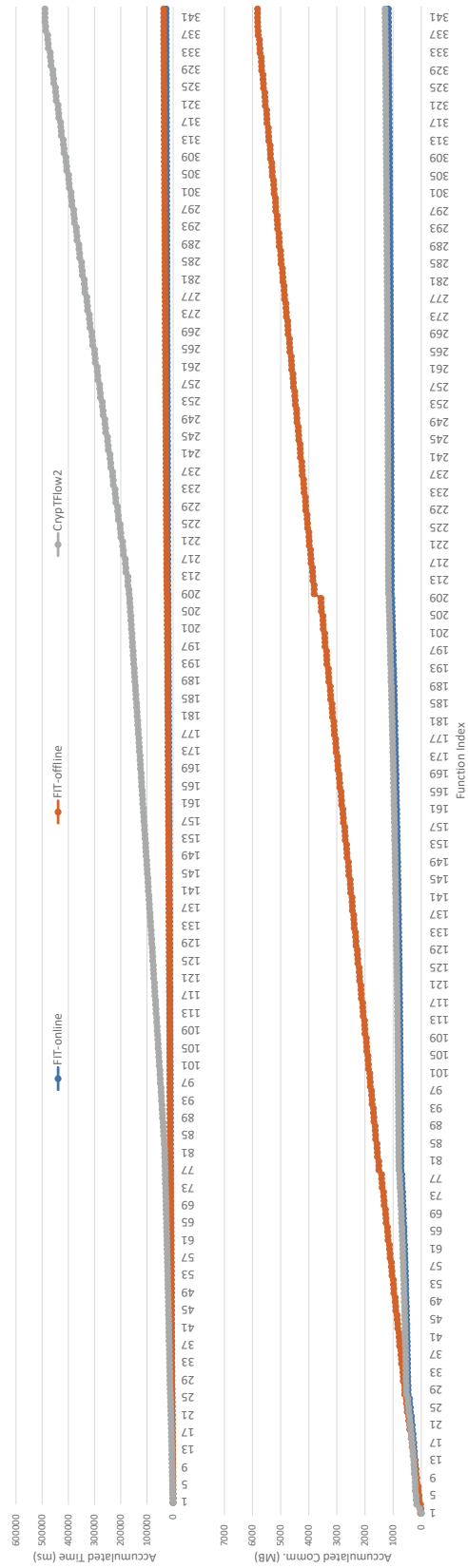

Figure 14. Performance Breakdown of DenseNet-169 with CIFAR10.

where the input channels and output channels are larger, the input-independent data transmission provides a mitigation for FIT's adaptability of privacy-preserving computation towards deeper networks.

Overall, the computation module of FIT, together with its network adaptation strategies, are applicable for various neural models with various input scales. Furthermore, the input-channel-dependent online computation makes FIT more appealing to be applied in large-size model with large-scale input as indicated in Table 2. This is because the increased number of output channels namely larger C_o leads to more speedup, while decreased number of channels in a ciphertext namely smaller C_n has negligible impact on the efficiency harvest.

5. Conclusion and Discussion

In this paper, we have looked back to the necessity of the function-wise methodology in state-of-the-art frameworks, and have initialized the formal investigation towards computing composite function for efficient privacy-preserving MLaaS. Under such fresh perspective, we have proposed FIT which features by a computation module for composite function with a series of joint optimization strategies. FIT remodels the process from function wise to allied counterpart that is from one function's input associated with the start of expensive overhead to another function's output enabling effective circumvention of unnecessary cost within the procedure. Such methodology has resulted in significant reduction of expensive overhead at running time. Theoretically, FIT not only eliminates the most expensive crypto operations without invoking extra encryption enabler, but also makes the running-time crypto complexity independent of filter size. Experimentally, FIT has demonstrated tens of times speedup over various function dimensions from modern networks, and 4.5× to 35.5× speedup for the total computation time when plugged in neural networks with data from small-scale MNIST to large-scale ImageNet.

While we have mainly investigated the performance of FIT over various CNN models with ReLU activation function, FIT can be readily integrated into other complex network frameworks as well as other activation functions. Specifically, in the popular network structures such as transformers, two of the main blocks are attention block and feed-forward block. While FIT is directly applicable to feed-forward block where the form is in $f_{\mathsf{w}}(f_{\mathsf{r}}(f_{\mathsf{w}}(x)))$, the attention block is in form of $f_{\mathsf{w}}(f_{\mathsf{s}}(f_{\mathsf{w}}(x)))$ where $f_{\mathsf{s}}(\cdot)$ is the softmax function and it could give us an opportunity to jointly optimize that block as long as the sum of exponential value in $f_{\mathsf{s}}(\cdot)$ can be transformed into comparison-based function. This is because FIT is directly applicable to comparison-based functions such as leakyReLU and piecewise-linear activation functions which are in form of Eq. (1). On the other hand, recent work [48] has shown that exponential activation functions such as SiLU, GeLU, and Mish can be approximated into piece-wise ones which provide us promising chances to adapt FIT to more complex activations as well as attention blocks in transformers.

Last but not least, although FIT is designed under semi-honest adversaries, it is adaptable to address security against a malicious adversary based on several existing techniques such as the mix-and-check approach [49]. The basic idea is that the client mixes the public samples with their own samples to be queried as the inputs to jointly perform the secure inference by calling FIT. If the server uses a low-quality model or deviates from the protocol, the client can easily identify it based on those public samples.

Acknowledgments

The research of Q. Zhang and T. Xiang is supported by the National Key R&D Program of China under Grant 2022YFB3103500, the National Natural Science Foundation of China under Grants 62302067, 62072062 and U20A20176, China Postdoctoral Science Foundation under Grant 2023M730407, the Fundamental Research Funds for the Central Universities under Grant 2022CDJXY-020, CCF-AFSG Research Fund under Grant RF20220009, and the Technology Innovation and Application Development Key Project supported by Chongqing Science and Technology Bureau under Grant CSTB2022TIAD-KPX0178.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[6] Y. Zhang, J. Qin, D. S. Park, W. Han, C.-C. Chiu, R. Pang, Q. V. Le, and Y. Wu, "Pushing the limits of semi-supervised learning for automatic speech recognition," *arXiv preprint arXiv:2010.10504*, 2020.

[7] S. Sohangir, D. Wang, A. Pomeranets, and T. M. Khoshgoftaar, "Big data: Deep learning for financial sentiment analysis," *Journal of Big Data*, vol. 5, no. 1, pp. 1–25, 2018.

[8] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of big data*, vol. 2, no. 1, pp. 1–21, 2015.

[9] H. C. Assistance, "Summary of the HIPAA privacy rule," *Office for Civil Rights*, 2003.

[10] M. Goddard, "The EU general data protection regulation (GDPR): European regulation that has a global impact," *International Journal of Market Research*, vol. 59, no. 6, pp. 703–705, 2017.

[11] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24*, 2016, pp. 201–210.

[12] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 619–631.

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

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

[15] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "XONN: Xnor-based oblivious deep neural network inference," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1501–1518.

[16] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "DELPHI: A cryptographic inference service for neural networks," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2505–2522.

[17] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.

[18] Q. Zhang, C. Xin, and H. Wu, "GALA: Greedy computation for linear algebra in privacy-preserved neural networks," in *ISOC Network and Distributed System Security Symposium*, 2021.

[19] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved mixed-protocol secure two-party computation," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[20] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "CRYPTGPU: Fast privacy-preserving machine learning on the gpu," *arXiv preprint arXiv:2104.10949*, 2021.

[21] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 707–721.

[22] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: Scalable provably-secure deep learning," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[23] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 35–52.

[24] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: a mixed-protocol machine learning framework for private inference," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[25] D. Demmler, T. Schneider, and M. Zohner, "ABY-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*, 2015.

[26] S. U. Hussain, M. Javaheripi, M. Samragh, and F. Koushanfar, "Coinn: Crypto/ml codesign for oblivious inference via neural networks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3266–3281.

[27] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *31th USENIX Security Symposium (USENIX Security 22)*, 2022.

[28] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*. Springer, 2012, pp. 868–886.

[29] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[30] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[31] G. Brassard, C. Crépeau, and J.-M. Robert, "All-or-nothing disclosure of secrets," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 234–238.

[32] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[33] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 784–796.

[34] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 1986, pp. 162–167.

[35] Q. Zhang, T. Xiang, C. Xin, B. Chen, and H. Wu, "Joint linear and nonlinear computation across functions for efficient privacy-preserving neural network inference," *arXiv preprint arXiv:2209.01637*, 2022.

[36] S. Halevi and V. Shoup, "Algorithms in helib," in *CRYPTO*, 2014, pp. 554–571.

[37] Y. Lindell, "How to simulate it - a tutorial on the simulation proof technique," *Cryptology ePrint Archive, Report 2016/046*, 2016.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[39] S. Li, K. Xue, B. Zhu, C. Ding, X. Gao, D. Wei, and T. Wan, "Falcon: A fourier transform based approach for fast and secure convolutional neural network predictions," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 8705–8714.

[40] X. Liu, Y. Zheng, X. Yuan, and X. Yi, "Securely outsourcing neural network inference to the cloud with lightweight techniques," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[41] https://github.com/tensorflow/tensorflow/issues/3180, 2022.

[42] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 192–206.

[43] "MNIST," http://yann.lecun.com/exdb/mnist/, 2021.

[44] S. An, M. Lee, S. Park, H. Yang, and J. So, "An ensemble of simple convolutional neural network models for mnist digit recognition," 2020.

[45] "CIFAR10," https://www.cs.toronto.edu/ kriz/cifar.html, 2021.

[46] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "Imagenet: A large-scale hierarchical image database," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, 2009, pp. 248–255.

[47] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," *arXiv:1608.06993*, 2018.

[48] M. Islam, S. S. Arora, R. Chatterjee, P. Rindal, and M. Shirvanian, "Compact: Approximating complex activation functions for secure computation," *CoRR*, vol. abs/2309.04664, 2023.

[49] C. Dong, J. Weng, J.-N. Liu, Y. Zhang, Y. Tong, A. Yang, Y. Cheng, and S. Hu, "Fusion: Efficient and secure inference resilient to malicious servers," in *Proceedings of the 2023 Network and Distributed System Security (NDSS) Symposium*, 2023, pp. 1–18.

[50] https://github.com/huyvnphan/PyTorch_CIFAR10, 2023.

[51] https://pytorch.org/vision/stable/models.html, 2023.

[52] https://github.com/aaron-xichen/pytorch-playground/, 2023.

Appendix A. Model Configurations

The M1 model is trained using the M_7 architecture in [44]. It contains four convolutions each of which is followed by batch normalization and ReLU, and one fully connection. The Adam optimizer is adopted with weight decay and learning rate equal to 0.0001 and 0.01, respectively. Meanwhile, the learning rate is lowered by 10 times at epoch 60 and 100. The batch size is set to 64 and the number of epochs is 200. As for the models with CIFAR10, we utilize the pretrained models vgg16_bn, vgg19_bn, resnet18, resnet34, densenet161 and densenet169 from [50]. As for the models with ImageNet, we use the pretrained models vgg16_bn, vgg19, resnet18, resnet34 from torchvision [51]. We quantize the above models based on methodology from [52] to deal with the limited plaintext space in FIT and the model accuracy is almost the same compared to the baseline as shown in Table 5.

TABLE 5. COMPARISON OF ACCURACY BEFORE AND AFTER QUANTIZATION (BEFORE IN % /AFTER IN %).

Models	Datasets		
	CIFAR10	ImageNet	MNIST
M1	-	-	99.4/99.5
VGG-16	94.0/93.9	71.8/71.8	-
VGG-19	93.9/93.9	70.9/70.9	-
ResNet-18	93.0/93.0	68.4/68.4	-
ResNet-34	93.3/93.3	72.3/72.3	-
DenseNet-161	94.0/93.9	-	-
DenseNet-169	94.0/93.9	-	-

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

In this paper, the authors remodel the computation process of the same function in mainstream works to the allied counterpart, from one function's input associated with the start of expensive overhead to another function's output, effectively circumventing unnecessary costs within the procedure.

B.2. Scientific Contributions

- Addresses a Long-Known Issue.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

1) The authors have proposed an interesting framework to improve the computational efficiency of privacy-preserving MLaaS. This framework also reduces the total computational cost compared with SOTA due to its unique design featuring joint optimization of composite functions.
2) In addition to the performance increase, this framework eliminates all rotations and has a running time independent of the convolutional layer filter size.