GAZELLE: A Low Latency Framework for Secure Neural Network Inference

Chiraag Juvekar, ***MIT MTL;*** **Vinod Vaikuntanathan,** ***MIT CSAIL;***
Anantha Chandrakasan, ***MIT MTL***

https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

GAZELLE: A Low Latency Framework for Secure Neural Network Inference

Chiraag Juvekar
MIT MTL

Vinod Vaikuntanathan
MIT CSAIL

Anantha Chandrakasan
MIT MTL

Abstract

The growing popularity of cloud-based machine learning raises natural questions about the privacy guarantees that can be provided in such settings. Our work tackles this problem in the context of *prediction-as-a-service* wherein a server has a convolutional neural network (CNN) trained on its private data and wishes to provide classifications on clients' private images. Our goal is to build efficient secure computation protocols which allow a client to obtain the classification result without revealing their input to the server, while at the same preserving the privacy of the server's neural network.

To this end, we design Gazelle, a scalable and low-latency system for secure neural network inference, using an intricate combination of homomorphic encryption and traditional two-party computation techniques (such as garbled circuits). Gazelle makes three contributions. First, we design a homomorphic encryption library which provides fast implementations of basic homomorphic operations such as SIMD (single instruction multiple data) addition, SIMD multiplication and ciphertext slot permutation. Second, we implement homomorphic linear algebra kernels which provide fast algorithms that map neural network layers to optimized homomorphic matrix-vector multiplication and convolution routines. Third, we design optimized encryption switching protocols which seamlessly convert between homomorphic and garbled circuit encodings to enable implementation of complete neural network inference.

We evaluate our protocols on benchmark neural networks trained on the MNIST and CIFAR-10 datasets and show that Gazelle outperforms the best existing systems such as MiniONN (ACM CCS 2017) and Chameleon (Crypto Eprint 2017/1164) by 20–30× in online runtime. When compared with fully homomorphic approaches like CryptoNets (ICML 2016), we demonstrate *three orders of magnitude* faster online run-time.

1 Introduction

Fueled by the massive influx of data, sophisticated algorithms and extensive computational resources, modern machine learning has found surprising applications in such diverse domains as medical diagnosis [43, 13], facial recognition [38] and credit risk assessment [2]. We consider the setting of supervised machine learning which proceeds in two phases: a *training* phase where a labeled dataset is turned into a model, and an *inference* or *classification* or *prediction* phase where the model is used to predict the label of a new unlabelled data point. Our work tackles a class of complex and powerful machine learning models, namely *convolutional neural networks* (CNN) which have demonstrated better-than-human accuracy across a variety of image classification tasks [28].

One important use-case for machine learning models (including CNNs) comes up in the setting of *predictions-as-a-service* (PaaS). In the PaaS setting, a large organization trains a machine learning model using its proprietary data. The organization now wants to monetize the model by deploying a service that allows clients to upload their inputs and receive predictions for a price.

The first solution that comes to mind is for the organization to make the model (in our setting, the architecture and parameters of the CNN) freely available for public consumption. This is undesirable for at least two reasons: first, once the model is given away, there is clearly no opportunity for the organization to monetize it, potentially removing its incentives to undergo the expensive data curating, cleaning and training phases; and secondly, the model, which has been trained on private organizational data, may reveal information about users that contributed to the dataset, violating their privacy and perhaps even regulations such as HIPAA.

A second solution that comes to mind is for the organization to build a web service that hosts the model and provides predictions for a small fee. However, this is also undesirable for at least two reasons: first, the users of such a service will rightfully be concerned about the privacy of the inputs they are providing to the web service; and secondly, the organization may not even want to know the user inputs for reasons of legal liability in case of a future data breach.

The goal of our work is to provide practical solutions to this conundrum of *secure neural network inference*. More concretely, we aim to enable the organization and its users to interact in such a way that the user eventually obtains the prediction (without learning the model) and the organization obtains no information about the user's input.

Modern cryptography provides us with many tools, such as fully homomorphic encryption and garbled circuits, that can help us solve this problem. A key take-away from our work is that both techniques have their limitations; understanding their precise trade-offs and using a combination of them judiciously in an application-specific manner helps us overcome the individual limitations and achieve substantial gains in performance. Indeed, several recent works [30, 36, 29, 18, 32] have built systems that address the problem of secure neural network inference using these cryptographic tools, and our work improves on all of them.

Let us begin by discussing these two techniques and their relative merits and shortcomings.

Homomorphic Encryption. Fully Homomorphic Encryption (FHE), is an encryption method that allows anyone to compute an arbitrary function f on an encryption of x, without decrypting it and without knowledge of the private key [34, 15, 6]. Using just the encryption of x, one can obtain an encryption of $f(x)$. Weaker versions of FHE, collectively called partially homomorphic encryption, permit the computation of a subset of all functions, typically functions that perform only additions (AHE) [31] or functions that can be computed by depth-bounded arithmetic circuits (LHE) [5, 4, 14]. Recent efforts, both in theory and in practice have given us large gains in the performance of several types of homomorphic schemes [5, 16, 7, 21, 35, 8] allowing us to implement a larger class of applications with better security guarantees.

The major bottleneck for these techniques, notwithstanding these recent developments, is their *computational complexity*. The computational cost of LHE, for example, grows dramatically with the depth of the circuit that the scheme needs to support. Indeed, the recent *CryptoNets* system gives us a protocol for secure neural network inference using LHE [18]. Largely due to its use of LHE, CryptoNets has two shortcomings. First, they need to change the structure of neural networks and retrain them with special LHE-friendly non-linear activation functions such as the square function. This has a potentially negative effect on the accuracy of these models. Secondly, and perhaps more importantly, even with these changes, the computational cost is prohibitively large. For example, on a neural network trained on the MNIST dataset, the end-to-end latency of CryptoNets is 297.5 *seconds*, in stark contrast to the 30 *milliseconds* end-to-end latency of Gazelle. In spite of the use of interaction, our online bandwidth per inference for this network is a mere 0.05MB as opposed to the 372MB required by CryptoNets.

In contrast to the LHE scheme in CryptoNets, Gazelle employs a much simpler *packed additively homomorphic encryption* (PAHE) scheme, which we show can support very fast matrix-vector multiplications and convolutions. Lattice-based AHE schemes come with powerful features such as SIMD evaluation and automorphisms (described in detail in Section 3) which make them the ideal tools for common linear-algebraic computations.

Secret Sharing and Garbled Circuits. Yao's garbled circuits [44] and the secret-sharing based Goldreich-Micali-Wigderson (GMW) protocol [19] are two leading methods for the task of two-party secure computation (2PC). After three decades of theoretical and applied work improving and optimizing these protocols, we now have very efficient implementations, e.g., [10, 9, 12, 33]. The modern versions of these techniques have the advantage of being computationally inexpensive, partly because they rely on symmetric-key cryptographic primitives such as AES and SHA and use them in a clever way [3], because of hardware support in the form of the Intel AES-NI instruction set, and because of techniques such as oblivious transfer extension [27, 3] which limit the use of public-key cryptography to an offline reusable pre-processing phase.

The major bottleneck for these techniques is their *communication complexity*. Indeed, three recent works followed the garbled circuits paradigm and designed systems for secure neural network inference: the SecureML system [30], the MiniONN system [29], the DeepSecure system [36].

DeepSecure uses garbled circuits alone; SecureML uses Paillier's AHE scheme to speed up some operations; and MiniONN uses a weak form of lattice-based AHE to generate "multiplication triples" similar to the SPDZ multiparty computation framework [9]. Our key claim is that understanding the precise trade-off point between AHE and garbled circuit-type techniques allows us to make optimal use of both and achieve large net computational and communication gains. In particular, in Gazelle, we use optimized AHE schemes in a completely different way from MiniONN: while they employ AHE as a pre-processing tool for generating triples, we use AHE to dramatically speed up linear algebra directly.

For example, on a neural network trained on the CIFAR-10 dataset, the most efficient of these three protocols, namely MiniONN, has an online bandwidth cost of 6.2GB whereas Gazelle has an online bandwidth cost of 0.3GB. In fact, we observe across the board a reduction of 20-80$\times$ in the online bandwidth per inference which gets better as the networks grow in size. In the LAN setting, this translates to an end-to-end latency of 3.6s versus the 72s for MiniONN.

Even when comparing to systems such as Chameleon [32] that rely on trusted third-party dealers, we observe a 30$\times$ reduction in online run-time and 2.5$\times$ reduction in online bandwidth, while simultaneously providing a pure two-party solution. A more detailed performance comparison with all these systems, is presented in Section 8.

(F)HE or Garbled Circuits? To use (F)HE and garbled circuits optimally, we need to understand the precise computational and communication trade-offs between them. Roughly speaking, homomorphic encryption performs better than garbled circuits when (a) the computation has small multiplicative depth, (ideally multiplicative depth 0 meaning that we are computing a linear function) and (b) the boolean circuit that performs the computation has large size, say quadratic in the input size. Matrix-vector multiplication (namely, the operation of multiplying a plaintext matrix with an encrypted vector) provides us with exactly such a scenario. Furthermore, the most time-consuming computations in a convolutional neural network are indeed the convolutional layers (which are

nothing but a special type of matrix-vector multiplication). The non-linear computations in a CNN such as the ReLU or MaxPool functions can be written as simple *linear-size* circuits which are best computed using garbled circuits. This analysis is the guiding philosophy that enables the design of Gazelle (A more detailed description of convolutional neural networks, is presented in Section 2).

Our System: The main contribution of this work is Gazelle, a framework for secure evaluation of convolutional neural networks. It consists of three components: The first component is the *Gazelle Homomorphic Layer* which consists of very fast implementations of three basic homomorphic operations: SIMD addition, SIMD scalar multiplication, and automorphisms (For a detailed description of these operations, see Section 3). Our innovations in this part consist of techniques for division-free arithmetic and techniques for lazy modular reductions. In fact, our implementation of the first two of these homomorphic operations is only 10-20× slower than the corresponding operations on plaintext.

The second component is the *Gazelle Linear Algebra kernels* which consists of very fast algorithms for homomorphic matrix-vector multiplications and homomorphic convolutions, accompanied by matching implementations. In terms of the basic homomorphic operations, SIMD additions and multiplications turn out to be relatively cheap whereas automorphisms are very expensive. At a very high level, our innovations in this part consists of several new algorithms for homomorphic matrix-vector multiplication and convolutions that minimize the expensive automorphism operations.

The third and final component is *Gazelle Network Inference* which uses a judicious combination of garbled circuits together with our linear algebra kernels to construct a protocol for secure neural network inference. Our innovations in this part consist of efficient protocols that switch between secret-sharing and homomorphic representations of the intermediate results and a novel protocol to ensure circuit privacy.

Our protocol also hides strictly more information about the neural network than other recent works such as the MiniONN protocol. We refer the reader to Section 2 for more details.

2 Secure Neural Network Inference

The goal of this section is to describe a clean abstraction of *convolutional neural networks* (CNN) and set up the secure neural inference problem that we will tackle in the rest of the paper. A CNN takes an input and processes it through a sequence of *linear* and *non-linear* layers in order to classify it into one of the potential classes. An example CNN is shown is Figure 1.

2.1 Linear Layers

The linear layers, shown in Figure 1 in red, can be of two types: convolutional (Conv) layers or fully-connected (FC) layers.

Convolutional Layers. We represent the input to a Conv layer by the tuple (w_i,h_i,c_i) where w_i is the image width, h_i is the image height, and c_i is the number of input channels. In other words, the input consists of c_i many $w_i \times h_i$ images. The convolutional layer is then parameterized by c_o filter banks each consisting of c_i many $f_w \times f_h$ filters. This is represented in short by the tuple (f_w,f_h,c_i,c_o). The computation in a Conv layer can be better understood in terms of simpler single-input single-output (SISO) convolutions. Every pixel in the output of a SISO convolution is computed by stepping a single $f_w \times f_h$ filter across the input image as shown in Figure 2. The output of the full Conv layer can then be parameterized by the tuple (w_o,h_o,c_o) which represents c_o many $w_o \times h_o$ output images. Each of these images is associated with a unique filter bank and is computed by the following two-step process shown in Figure 2: (i) For each of the c_i filters in the associated filter bank, compute a SISO convolution with the corresponding channel in the input image, resulting in c_i many intermediate images; and (ii) summing up all these c_i intermediate images.

There are two commonly used padding schemes when performing convolutions. In the *valid* scheme, no input padding is used, resulting in an output image that is smaller than the initial input. In particular we have $w_o = w_i - f_w + 1$ and $h_o = h_i - f_h + 1$. In the *same* scheme, the input is zero padded such that output image size is the same as the input.

In practice, the Conv layers sometimes also specify an additional pair of stride parameters (s_w, s_h) which denotes the granularity at which the filter is stepped. After accounting for the strides, the output image size (w_o,h_o), is given by $(\lfloor (w_i - f_w + 1)/s_w \rfloor, \lfloor (h_i - f_h + 1)/s_h \rfloor)$ for *valid* style convolutions and $(\lfloor w_i/s_w \rfloor, \lfloor h_i/s_h \rfloor)$ for *same* style convolutions.

Fully-Connected Layers. The input to a FC layer is a vector $\mathbf{v}_i$ of length n_i and its output is a vector $\mathbf{v}_o$ of length n_o. A fully connected layer is specified by the tuple $(\mathbf{W}, \mathbf{b})$ where $\mathbf{W}$ is $(n_o \times n_i)$ weight matrix and $\mathbf{b}$ is an n_o element bias vector. The output is specified by the following transformation: $\mathbf{v}_o = \mathbf{W} \cdot \mathbf{v}_i + \mathbf{b}$.

The key observation that we wish to make is that the number of multiplications in the Conv and FC layers are given by $(w_o \cdot h_o \cdot c_o \cdot f_w \cdot f_h \cdot c_i)$ and $n_i \cdot n_o$, respectively. This makes both the Conv and FC layer computations quadratic in the input size. This fact guides us to use homomorphic encryption rather than garbled circuit-based techniques to compute the convolution and fully connected layers, and indeed, this insight is at the heart of the much of the speedup achieved by Gazelle.

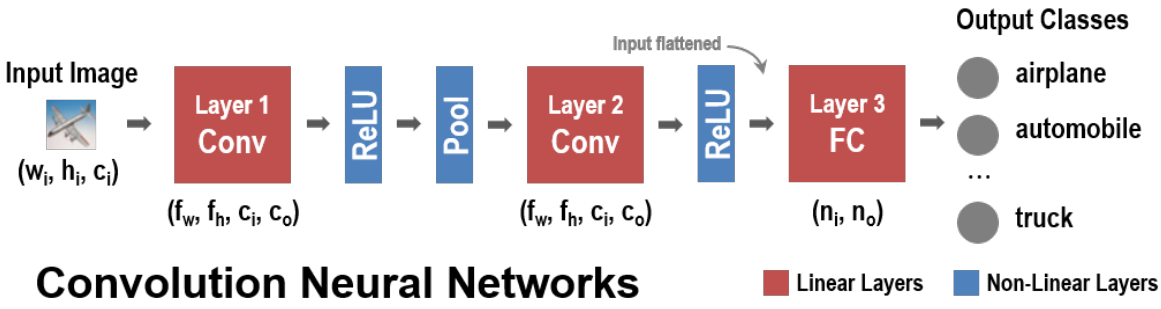

Figure 1: A CNN with two Conv layers and one FC layer. ReLU is used as the activation function and a MaxPooling layer is added after the first Conv layer.

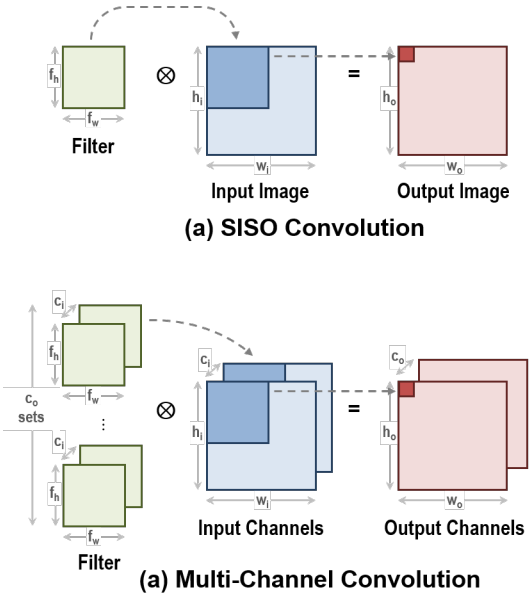

Figure 2: SISO convolutions and multi-channel Conv layers

2.2 Non-Linear Layers

The non-linear layers, shown in Figure 1 in blue, consist of an activation function that acts on each element of the input separately or a pooling function that reduces the output size. Typical non-linear functions can be one of several types: the most common in the convolutional setting are max-pooling functions and ReLU functions.

The key observation that we wish to make in this context is that all these functions can be implemented by circuits that have size linear in the input size and thus, evaluating them using conventional 2PC approaches does not impose any additional asymptotic communication penalty.

For more details on CNNs, we refer the reader to [40].

2.3 Secure Inference: Problem Description

In our setting, there are two parties A and B where A holds a convolutional neural network (CNN) and B holds an input to the network, typically an image. We make a distinction between the *structure* of the CNN which includes the number of layers, the size of each layer, and the activation functions applied in layer, versus the *parameters* of the CNN which includes all the weights and biases that describe the convolution and the fully connected layers.

We wish to design a protocol that A and B engage in at the end of which B obtains the classification result (and potentially the network structure), namely the output of the final layer of the neural network, whereas A obtains nothing.

The Threat Model. Our threat model is the same as in previous works, namely the SecureML, MiniONN and DeepSecure systems and our techniques, as we argue below, leak even less information than in these works.

To be more precise, we consider semi-honest corruptions as in [36, 29, 30], i.e., A and B adhere to the software that describes the protocol, but attempt to infer information about the other party's input (the network parameters or the image, respectively) from the protocol transcript. We ask for the cryptographic standard of ideal/real security [20, 19]. Two comments are in order about this ideal functionality.

The first is an issue specific to the ideal functionality instantiated in this and past work, i.e., the ideal functionality does not completely hide the *network structure*. We argue, however, that it does hide the important aspects which are likely to be proprietary. In particular, the ideal functionality and our realization hides all the weights and biases in the convolution and the fully connected layers. Secondly, we also hide the filter and stride size in the convolution layers, as well as information as to which layers are convolutional layers and which are fully connected. We do reveal the number of layers and the size[1] (the

[1] One can potentially hide this information by padding the network with dummy operation at a proportional computational expense

number of hidden nodes) of each layer. In contrast, other protocols for secure neural network inference such as the MiniONN protocol [29] reveal strictly more information, e.g., they reveal the filter size. As for party B's security, we hide the entire image, but not its size, from party A.

A second, more subtle, issue is with the definition of the ideal functionality which implements secure network inference. Since such functionality, must at a bare minimum, give B access to the classification output, B maybe be able to train a new classifier to mimic these classification results. This attack is called model stealing [42]. Note that model stealing with limited queries is essentially equivalent to a supervised learning task with access to a limited training dataset. Thus a potential model stealing adversary could train such classifier without access to B by simply asking a domain expert to classify his limited set of test-images. One potential solution is to limit the number of classification queries that A is allowed to make of the model. This can be a practical solution in a *try-before-buy* scenario where B only needs access to limited set of classifications to test the performance of the network before it buy the network parameters from A. We remark that designing (potentially-noisy) classifiers which are intrinsically resilient to model stealing is an interesting open machine learning problem.

Paper Organization. The rest of the paper is organized as follows. We first describe our abstraction of a packed additively homomorphic encryption (PAHE) that we use through the rest of the paper. We then provide an overview of the entire Gazelle protocol in section 4. In the next two sections, Section 5 and 6, we elucidate the most important technical contributions of the paper, namely the *linear algebra kernels* for fast matrix-vector multiplication and convolution. We then present detailed benchmarks on the implementation of the *homomorphic encryption layer* and the linear algebra kernels in Section 7. Finally, we describe the evaluation of neural networks such as ones trained on the MNIST or CIFAR-10 datasets and compare Gazelle's performance to prior work in Section 8.

3 Packed Additively Homomorphic Encryption

In this section, we describe a clean abstraction of packed additively homomorphic encryption (PAHE) schemes that we will use through the rest of the paper. As suggested by the name, the abstraction will support packing multiple plaintexts into a single ciphertext, performing SIMD homomorphic additions (SIMDAdd) and scalar multiplications (SIMDScMult), and permuting the plaintext slots (Perm). In particular, we will never need or use homomorphic multiplication of two ciphertexts. This abstraction can be instantiated with essentially all modern lattice-based homomorphic encryption schemes, e.g., [5, 16, 4, 14].

For the purposes of this paper, a private-key PAHE suffices. In such an encryption scheme, we have a (randomized) encryption algorithm (PAHE.Enc) that takes a plaintext message vector $\mathbf{u}$ from some message space and encrypts it using a key sk into a ciphertext denoted as $[\mathbf{u}]$, and a (deterministic) decryption algorithm (PAHE.Dec) that takes the ciphertext $[\mathbf{u}]$ and the key sk and recovers the message $\mathbf{u}$. Finally, we also have a homomorphic evaluation algorithm (PAHE.Eval) that takes as input one or more ciphertexts that encrypt messages $M_0, M_1, \ldots$, and outputs another ciphertext that encrypts a message $M = f(M_0, M_1, \ldots)$ for some function f constructed using the SIMDAdd, SIMDScMult and Perm operations. We require IND-CPA security, which requires that ciphertexts of any two messages $\mathbf{u}$ and $\mathbf{u}'$ be computationally indistinguishable.

The lattice-based PAHE constructions that we consider in this paper are parameterized by four constants: (1) the cyclotomic order m, (2) the ciphertext modulus q, (3) the plaintext modulus p and (4) the standard deviation σ of a symmetric discrete Gaussian noise distribution (χ).

The number of slots in a packed PAHE ciphertext is given by $n = \phi(m)$ where ϕ is the Euler Totient function. Thus, plaintexts can be viewed as length-n vectors over $\mathbb{Z}_p$ and ciphertexts are viewed as length-n vectors over $\mathbb{Z}_q$. All fresh ciphertexts start with an inherent noise η sampled from the noise distribution χ. As homomorphic computations are performed η grows continually. Correctness of PAHE.Dec is predicated on the fact that $|\eta| < q/(2p)$, thus setting an upper bound on the complexity of the possible computations.

In order to guarantee security we require a minimum value of σ (based on q and n), $q \equiv 1 \bmod m$ and p is co-prime to q. Additionally, in order to minimize noise growth in the homomorphic operations we require that the magnitude of $r \equiv q \bmod p$ be as small as possible. This when combined with the security constraint results in an optimal value of $r = \pm 1$.

In the sequel, we describe in detail the three basic operations supported by the homomorphic encryption schemes together with their associated asymptotic cost in terms of (a) the run-time, and (b) the noise growth. Later, in Section 7, we will provide concrete micro-benchmarks for each of these operations implemented in the GAZELLE library.

3.1 Addition: SIMDAdd

Given ciphertexts $[\mathbf{u}]$ and $[\mathbf{v}]$, SIMDAdd outputs an encryption of their component-wise sum, namely $[\mathbf{u}+\mathbf{v}]$.

The asymptotic run-time for homomorphic addition is $n \cdot \mathsf{CostAdd}(q)$, where $\mathsf{CostAdd}(q)$ is the run-time for adding two numbers in $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$. The noise growth is at most $\eta_{\mathbf{u}} + \eta_{\mathbf{v}}$ where $\eta_{\mathbf{u}}$ (resp. $\eta_{\mathbf{v}}$) is the amount of noise in $[\mathbf{u}]$ (resp. in $[\mathbf{v}]$).

3.2 Scalar Multiplication: SIMDScMult

If the plaintext modulus is chosen such that $p \equiv 1 \bmod m$, we can also support a SIMD compenentwise product.

Thus given a ciphertext $[\mathbf{u}]$ and a plaintext $\mathbf{v}$, we can output an encryption $[\mathbf{u} \circ \mathbf{v}]$ (where $\circ$ denotes component-wise multiplication of vectors).

The asymptotic run-time for homomorphic scalar multiplication is $n \cdot \mathsf{CostMult}(q)$, where $\mathsf{CostMult}(q)$ is the run-time for multiplying two numbers in $\mathbb{Z}_q$. The noise growth is at most $\eta_{\mathsf{mult}} \cdot \eta_{\mathbf{u}}$ where $\eta_{\mathsf{mult}} \approx ||\mathbf{v}||'_\infty \cdot \sqrt{n}$ is the *multiplicative noise growth* of the SIMD scalar multiplication operation.

For a reader familiar with homomorphic encryption schemes, we note that $||\mathbf{v}||'_\infty$ is the largest value in the *coefficient representation* of the packed plaintext vector $\mathbf{v}$, and thus, even a binary plaintext vector can result in η_{mult} as high as $p \cdot \sqrt{n}$. In practice, we alleviate this large multiplicative noise growth by bit-decomposing the coefficient representation of $\mathbf{v}$ into $\log(p/2^{\mathsf{w_{pt}}})$ many $\mathsf{w_{pt}}$-sized chunks $\mathbf{v}_k$ such that $\mathbf{v} = \sum 2^{\mathsf{w_{pt}} \cdot k} \cdot \mathbf{v}_k$. We refer to $\mathsf{w_{pt}}$ as the plaintext window size.

We can now represent the product $[\mathbf{u} \circ \mathbf{v}]$ as $\sum [\mathbf{u}_k \circ \mathbf{v}_k]$ where $\mathbf{u}_k = [2^{\mathsf{w_{pt}} \cdot k} \cdot \mathbf{u}]$. Since $||\mathbf{v}_k||'_\infty \leq 2^{\mathsf{w_{pt}}}$ the total noise in the multiplication is bounded by $2^{\mathsf{w_{pt}}} \cdot k\sqrt{n} \cdot \eta_{\mathbf{u}_k}$ as opposed to $p \cdot \sqrt{n} \cdot \eta_{\mathbf{u}}$. The only caveat is that we need access to low noise encryptions $[\mathbf{u}_k]$ as opposed to just $[\mathbf{u}]$ as in the direct approach.

3.3 Slot Permutation: Perm

Given a ciphertext $[\mathbf{u}]$ and one of a set of *primitive permutations* π defined by the scheme, the Perm operation outputs a ciphertext $[\mathbf{u}_\pi]$, where $\mathbf{u}_\pi$ is defined as $(u_{\pi(1)}, u_{\pi(2)}, \ldots, u_{\pi(n)})$, namely the vector $\mathbf{u}$ whose slots are permuted according to the permutation π. The set of permutations that can be supported depends on the structure of the multiplicative group $\bmod m$ i.e. $(\mathbb{Z}/m\mathbb{Z})^\times$. When m is prime, we have $n\,(=m-1)$ slots and the permutation group supports all cyclic rotations of the slots, i.e. it is isomorphic to C_n (the cyclic group of order n). When m is a sufficiently large power of two $(m = 2^k, m \geq 8)$, we have $n = 2^{k-1}$ and the set of permutations is isomorphic to the set of half-rotations i.e. $C_{n/2} \times C_2$, as illustrated in Figure 4.

Permutations are by far the most expensive operations in a homomorphic encryption scheme. At a high-level the PAHE ciphertext vectors represent polynomials. The permutation operation requires transforming these polynomials from evaluation to coefficient representations and back. These transformations can be efficiently computed using the number theoretic transform (NTT) and its inverse, both of which are finite-field analogues of their real valued Discrete Fourier Transform counterparts. Both the NTT and NTT^{-1} have an asymptotic cost of $\Theta(n \log n)$. As shown in [6], we need to perform $\Theta(\log q)$ NTT^{-1} to control Perm noise growth. The total cost of Perm is therefore $\Theta(n \log n \log q)$ operations. The noise growth is additive, namely, $\eta_{\mathbf{u}_\pi} = \eta_{\mathbf{u}} + \eta_{\mathsf{rot}}$ where η_{rot} is the *additive noise growth* of a permutation operation.

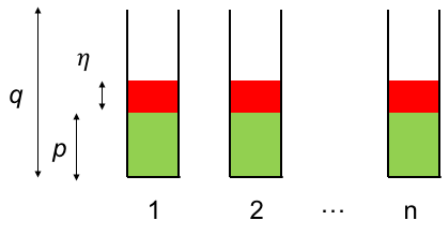

Figure 3: Ciphertext Structure and Operations. Here, n is the number of slots, q is the size of ciphertext space (so a ciphertext required $\lceil \log_2 q \rceil$ bits to represent), p is the size of the plaintext space (so a plaintext can have at most $\lfloor \log_2 p \rfloor$ bits), and η is the amount of noise in the ciphertext.

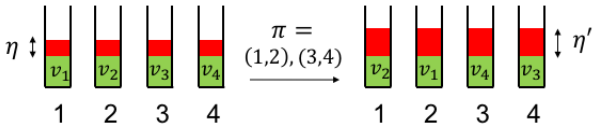

Figure 4: A Plaintext Permutation in action. The permutation π in this example swaps the first and the second slots, and also the third and fourth slots. The operation incurs a noise growth from η to $\eta' \approx \eta + \eta_{\mathsf{rot}}$. Here, $\eta_{\mathsf{rot}} \approx n \log q \cdot \eta_0$ where η_0 is some small "base noise".

3.4 Paillier vs. Lattice-based PAHE

The PAHE scheme used in Gazelle is dramatically more efficient than conventional Paillier based AHE. Homomorphic addition of two Paillier ciphertexts corresponds to a modular multiplication modulo a large RSA-like modulus (3072bits) as opposed to a simple addition mod q as seen in SIMDAdd. Similarly multiplication by a plaintext turns into a modular exponentiation for Paillier. Furthermore the large sizes of the Paillier ciphertexts makes encryption of single small integers extremely bandwidth-inefficient. In contrast, the notion of packing provided by lattice-based schemes provides us with a SIMD way of packing many integers into one ciphertext, as well as SIMD evaluation algorithms. We are aware of one system [37] that tries to use Paillier in a SIMD fashion; however, this lacks two crucial components of lattice-based AHE, namely the facility to multiply each slot with a separate scalar, and the facility to permute the slots. We are also aware of a method of mitigating the first of these shortcomings [26], but not the second. Our fast homomorphic implementation of linear algebra uses both these features of lattice-based AHE, making Paillier an inefficient substitute.

3.5 Parameter Selection for PAHE

Parameter selection for PAHE requires a delicate balance between the homomorphic evaluation capabilities and the target security level. We detail our procedure for parameter selection to meet a target security level of 128 bits. We first set our plaintext modulus to be 20 bits to represent the fixed point inputs (the bit-length of each pixel in an image) and partial sums generated during the neural network evaluation. Next, we require that the ciphertext modulus be close to, but less than, 64 bits in order to ensure that each ciphertext slot fits in a single machine word while maximizing the potential noise margin available during homomorphic computation.

The Perm operation in particular presents an interesting tradeoff between the simplicity of possible rotations and the computational efficiency of the NTT. A prime m results in a (simpler) cyclic permutation group but necessitates the use of an expensive Bluestein transform. Conversely, the use of $m = 2^k$ allows for a $8\times$ more efficient Cooley-Tukey style NTT at the cost of an awkward permutation group that only allows half-rotations. In this work, we opt for the latter and adapt our linear algebra kernels to deal with the structure of the permutation group. Based on the analysis of [1], we set $m = 4096$ and $\sigma = 4$ to obtain our desired security level.

Our chosen bit-width for q (60 bits), allows for lazy reduction, i.e. multiple additions may be performed without overflowing a machine word before a reduction is necessary. Additionally, even when q is close to the machine word-size, we can replace modular reduction with a simple sequence of addition, subtraction and multiplications. This is done by choosing q to be a pseudo-Mersenne number.

Next, we detail a technique to generate prime moduli that satisfy the above correctness and efficiency properties, namely:

1. $q \equiv 1 \bmod m$
2. $p \equiv 1 \bmod m$
3. $|q \bmod p| = |r| \approx 1$
4. q is pseudo-Mersenne, i.e. $q = 2^{60} - \delta, (\delta < \sqrt{q})$

Since we have chosen m to be a power of two, we observe that $\delta \equiv -1 \bmod m$. Moreover, $r \equiv q \bmod p$ implies that $\delta \equiv (2^{60} - r) \bmod p$. These two CRT expressions for δ imply that given a prime p and residue r, there exists a unique minimal value of $\delta \bmod (p \cdot m)$.

Based on this insight our prime selection procedure can be broken down into three steps:

1. Sample for $p \equiv 1 \bmod m$ and sieve the prime candidates.
2. For each candidate p, compute the potential $2|r|$ candidates for δ (and thus q).
3. If q is prime and δ is sufficiently small accept the pair (p,q).

Heuristically, this procedure needs $\log(q)(p \cdot m)/(2|r|\sqrt{q})$ candidate primes p to sieve out a suitable q.

Table 1: Prime Selection for PAHE

$\lfloor \log(p) \rfloor$	p	q	$\lvert r \rvert$
18	307201	$2^{60} - 2^{12} \cdot 63549 + 1$	1
22	5324801	$2^{60} - 2^{12} \cdot 122130 + 1$	1
26	115351553	$2^{60} - 2^{12} \cdot 9259 + 1$	1
30	1316638721	$2^{60} - 2^{12} \cdot 54778 + 1$	2

Since $p \approx 2^{20}$ and $q \approx 2^{64}$ in our setting, this procedure is very fast. A list of reduction-friendly primes generated by this approach is tabulated in Table 1. Finally note that when $\lfloor \log(p) \rfloor \cdot 3 < 64$ we can use Barrett reduction to speed-up reduction $\bmod p$.

The impact of the selection of reduction-friendly primes on the performance of the PAHE scheme is described in section 7.

4 Our Protocol at a High Level

Our protocol for secure neural network inference is based on the alternating use of PAHE and garbled circuits (GC). We will next explain the flow of the protocol and show how one can efficiently and securely convert between the data representations required for the two cryptographic primitives.

The main invariant that the protocol maintains is that at the start of the PAHE phase the server and the client posses an additive share c_y, s_y of the client's input $\mathbf{y}$. At the very beginning of the computation this can be accomplished by the trivial share $(c_y, s_y) = (\mathbf{y}, 0)$.

In order to evaluate a linear layer, we start with the client B first encrypting their share using the PAHE scheme and sending it to the server A. A in turn homomorphically adds her share s_y to obtain an encryption of $c_y + s_y = [\mathbf{y}]$. The security of the homomorphic encryption scheme guarantees that B cannot recover $\mathbf{y}$ from this encryption. The server A then uses a homomorphic linear algebra kernel to evaluate linear layer (which is either convolution or fully connected). The result is a packed ciphertext that contains the input to the first non-linear (ReLU) layer. The homomorphic scheme ensures that A learns nothing about B's input. B has not received any input from A yet and thus has no way of learning the model parameters.

In preparation for the evaluation of the subsequent non-linear activation layer A must transform her PAHE ciphertext into additive shares. At the start of this step A holds a ciphertext $[\mathbf{x}]$ (where $\mathbf{x}$ is a vector) and B holds the private key. The first step is to transform this ciphertext such that both A and B hold an additive secret sharing of $\mathbf{x}$. This is accomplished by the server A adding a random vector $\mathbf{r}$ to her ciphertext homomorphically to obtain an encryption $[\mathbf{x} + \mathbf{r}]$ and sending it to the client B. The client B then decrypts this message to get his share. Thus the server A sets her share $s_x = \mathbf{r}$ and B sets his share $c_x = \mathbf{x} + \mathbf{r} \bmod p$.

Since A chooses $\mathbf{r}$ uniformly at random s_x does not contain any information about either the model or B's input. Since B does not know $\mathbf{r}$, c_x has a uniform random distribution from B's perspective. Moreover the security of the PAHE scheme ensures that A has no way of figuring out what c_x is.

We next evaluate the non-linear activation using Yao's GC protocol. At the start of this step both parties posses additive shares (c_x, s_x) of the secret value of x and want to compute $y = \mathsf{ReLU}(x)$ without revealing it completely to either party. We evaluate the non-linear activation function ReLU (in parallel for each component of $\mathbf{x}$) to get a secret sharing of the output $\mathbf{y} = \mathsf{ReLU}(\mathbf{x})$. This is done using our circuit from Figure 5, described in more detail below. The output of the garbled circuit evaluation is a pair of shares s_y (for the server) and c_y (for the client) such that $s_y + c_y = y \bmod p$. The security argument is exactly the same as after the first step, i.e. neither party has complete information and both shares appear uniformly random to their respective owners.

Once this is done, we are back where we started and we can repeat these steps until we evaluate the full network. We make the following two observations about our proposed protocol:

1. By using AHE for the linear layers, we ensure that the communication complexity of protocol is linear in the number of layers and the size of inputs for each layer.
2. At the end of the garbled circuit protocol we have an additive share that can be encrypted afresh. As such, we can view the re-encryption as an interactive bootstrapping procedure that clears the noise introduced by any previous homomorphic operation.

For the second step of the outline above, we employ the *boolean* circuit described in Figure 5. The circuit takes as input three vectors: $s_x = \mathbf{r}$ and $s_y = \mathbf{r}'$ (chosen at random) from the server, and c_x from the client. The first block of the circuit computes the arithmetic sum of s_x and c_x over the integers and subtracts p from to obtain the result mod p. (The decision of whether to subtract p or not is made by the multiplexer). The second block of the circuit computes a ReLU function. The third block adds the result to s_y to obtain the client's share of y, namely c_y. For more detailed benchmarks on the ReLU and MaxPool garbled circuit implementations, we refer the reader to Section 8. We note that this conversion strategy is broadly similar to the one developed in [25].

In our evaluations, we consider ReLU, Max-Pool and the square activation functions, the first two are by far the most commonly used ones in convolutional neural network design [28, 41, 39, 24]. Note that the square activation function popularized for secure neural network evaluation in [18] can be efficiently implemented by a simple interactive protocol that uses the PAHE scheme to generate the cross-terms.

The use of an IND-CPA-secure PAHE scheme for evaluating the linear layers guarantees the privacy of the client's inputs. However the PAHE scheme must also guarantee the confidentiality of the server's input, in other words, it should be *circuit-private*. Prior work addresses this problem in two ways. The first approach called noise-flooding adds a large amount of noise to the final ciphertext [15] to obscure any information leaked through the ciphertext noise. The second technique relies on bootstrapping, either using garbled circuits [17] or using the full power of an FHE scheme [11]. Noise-flooding causes an undesirable blow-up in the parameters of the underlying PAHE scheme, while the FHE-bootstrapping based solution is well beyond the scope of the simple PAHE schemes we employ. Thus, our solution builds a low-overhead circuit-private interactive decryption protocol (Appendix B) to improve the concrete efficiency of the garbled circuit approach (as in [17]) as applied to the BFV scheme [4, 14].

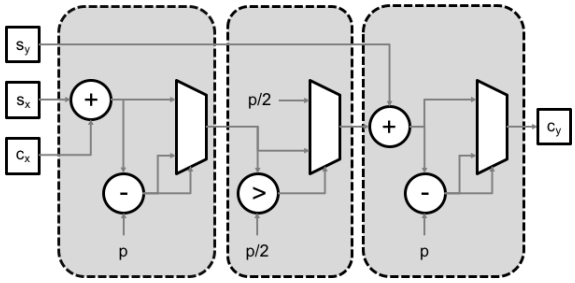

Figure 5: Our combined circuit for steps (a), (b) and (c) for the non-linear layers. The "+" gates refer to an integer addition circuit, "-" refers to an integer subtraction circuit and the ">" refers to the circuit refers to a greater than comparison. Note that the borrow of the subtraction gates is used as the select for the first and last multiplexer

5 Fast Homomorphic Matrix-Vector Multiplication

We next describe the homomorphic linear algebra kernels that compute matrix-vector products (for FC layers) and 2D convolutions (for Conv layers). In this section, we focus on matrix-vector product kernels which multiply a plaintext matrix with an encrypted vector. We start with the easiest to explain (but the slowest and most communication-inefficient) methods and move on to describing optimizations that make matrix-vector multiplication much faster. In particular, our *hybrid method* (see Table 4 and the description below) gives us the best performance among all our homomorphic matrix-vector multiplication methods. For example, multiplying a 128×1024 matrix with a length-1024 vector using our hybrid scheme takes about 16ms(For detailed benchmarks, we refer the reader to Section 7.3). In all the subsequent examples, we will use an FC layer with n_i inputs and

Table 2: Comparing matrix-vector product algorithms by operation count, noise growth and number of output ciphertexts

	Perm (Hoisted)[a]	Perm	SIMDScMult	SIMDAdd	Noise	#out_ct[b]
Naïve	0	$n_o \cdot \log n_i$	n_o	$n_o \cdot \log n_i$	$\eta_{\mathsf{naive}} := \eta_0 \cdot \eta_{\mathsf{mult}} \cdot n_i + \eta_{\mathsf{rot}} \cdot (n_i - 1)$	n_o
Naïve (Output packed)	0	$n_o \cdot \log n_i + n_o - 1$	$2 \cdot n_o$	$n_o \cdot \log n_i + n_o$	$\eta_{\mathsf{naive}} \cdot \eta_{\mathsf{mult}} \cdot n_o + \eta_{\mathsf{rot}} \cdot (n_o - 1)$	1
Naïve (Input packed)	0	$\frac{n_o \cdot n_i}{n} \cdot \log n_i$	$\frac{n_o \cdot n_i}{n}$	$\frac{n_o \cdot n_i}{n} \cdot \log n_i$	$\eta_0 \cdot \eta_{\mathsf{mult}} \cdot n_i + \eta_{\mathsf{rot}} \cdot (n_i - 1)$	$\frac{n_o \cdot n_i}{n}$
Diagonal	$n_i - 1$	0	n_i	n_i	$(\eta_0 + \eta_{\mathsf{rot}}) \cdot \eta_{\mathsf{mult}} \cdot n_i$	1
Hybrid	$\frac{n_o \cdot n_i}{n} - 1$	$\log \frac{n}{n_o}$	$\frac{n_o \cdot n_i}{n}$	$\frac{n_o \cdot n_i}{n} + \log \frac{n}{n_o}$	$(\eta_0 + \eta_{\mathsf{rot}}) \cdot \eta_{\mathsf{mult}} \cdot n_i + \eta_{\mathsf{rot}} \cdot (\frac{n_i}{n_o} - 1)$	1

[a] Rotations of the input with a common PermDecomp [b] Number of output ciphertexts
[c] All logarithms are to base 2

n_o outputs as a running example. For simplicity of presentation, unless stated otherwise we assume that n, n_i and n_o are powers of two. Similarly we assume that n_o and n_i are smaller than n. If not, we can split the original matrix into $n \times n$ sized blocks that are processed independently.

The Naïve Method. In the naïve method, each row of the $n_o \times n_i$ plaintext weight matrix $\mathbf{W}$ is encoded into a separate plaintext vectors (see Figure 6). Each such vector is of length n; where the first n_i entries contain the corresponding row of the matrix and the other entries are padded with 0. These plaintext vectors are denoted $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{(n_o-1)}$. We then use SIMDScMult to compute the component-wise product of with the encrypted input vector $[\mathbf{v}]$ to get $[\mathbf{u}_i] = [\mathbf{w}_i \circ \mathbf{v}]$. In order to compute the inner-product what we need is actually the sum of the entries in each of these vectors $\mathbf{u}_i$.

This can be achieved by a "rotate-and-sum" approach, where we first rotate the entries of $[\mathbf{u}_i]$ by $n_i/2$ positions. The result is a ciphertext whose first $n_i/2$ entries contain the sum of the first and second halves of $\mathbf{u}_i$. One can then repeat this process for $\log_2 n_i$ iterations, rotating by half the previous rotation on each iteration, to get a ciphertext whose first slot contains the first component of $\mathbf{Wv}$. By repeating this procedure for each of the n_o rows we get n_o ciphertexts, each containing one element of the result.

Based on this description, we can derive the following performance characteristics for the naïve method:

- The total cost is n_o SIMD scalar multiplications, $n_o \cdot \log_2 n$ rotations (automorphisms) and $n_o \cdot \log_2 n$ SIMD additions.
- The noise grows from η to $\eta \cdot \eta_{\mathsf{mult}} \cdot n + \eta_{\mathsf{rot}} \cdot (n-1)$ where η_{mult} is the multiplicative noise growth factor for SIMD multiplication and η_{rot} is the additive noise growth for a rotation. This is because the one SIMD multiplication turns the noise from $\eta \mapsto \eta \cdot \eta_{\mathsf{mult}}$, and the sequence of rotations and additions grows the noise as follows:
  $\eta \cdot \eta_{\mathsf{mult}} \mapsto (\eta \cdot \eta_{\mathsf{mult}}) \cdot 2 + \eta_{\mathsf{rot}} \mapsto (\eta \cdot \eta_{\mathsf{mult}}) \cdot 4 + \eta_{\mathsf{rot}} \cdot 3 \mapsto \ldots$
  which gives us the above result.
- Finally, this process produces n_o many ciphertexts each one containing just one component of the result.

This last fact turns out to be an unacceptable efficiency barrier. In particular, the total network bandwidth becomes quadratic in the input size and thus contradicts the entire rationale of using PAHE for linear algebra. Ideally, we want the entire result to come out in packed form *in a single ciphertext* (assuming, of course, that $n_o \leq n$).

A final subtle point that needs to noted is that if n is not a power of two, then we can continue to use the same rotations as before, but all slots except the first slot leak information about partial sums. We therefore *must* add a random number to these slots to destroy this extraneous information about the partial sums.

5.1 Output Packing

The very first thought to mitigate the ciphertext blowup issue we just encountered is to take the many output ciphertexts and somehow pack the results into one. Indeed, this can be done by (a) doing a SIMD scalar multiplication which zeroes out all but the first coordinate of each of the out ciphertexts; (b) rotating each of them by the appropriate amount so that the numbers are lined up in different slots; and (c) adding all of them together.

Unfortunately, this results in unacceptable noise growth. The underlying reason is that we need to perform two serial SIMD scalar multiplications (resulting in an η_{mult}^2 factor; see Table 4). For most practical settings, this noise growth forces us to use ciphertext moduli that are larger 64 bits, thus overflowing the machine word. This necessitates the use of a Double Chinese Remainder Theorem (DCRT) representation similar to [16] which substantially slows down computation. Instead we use an algorithmic approach to control noise growth allowing the use of smaller moduli and avoiding the need for DCRT.

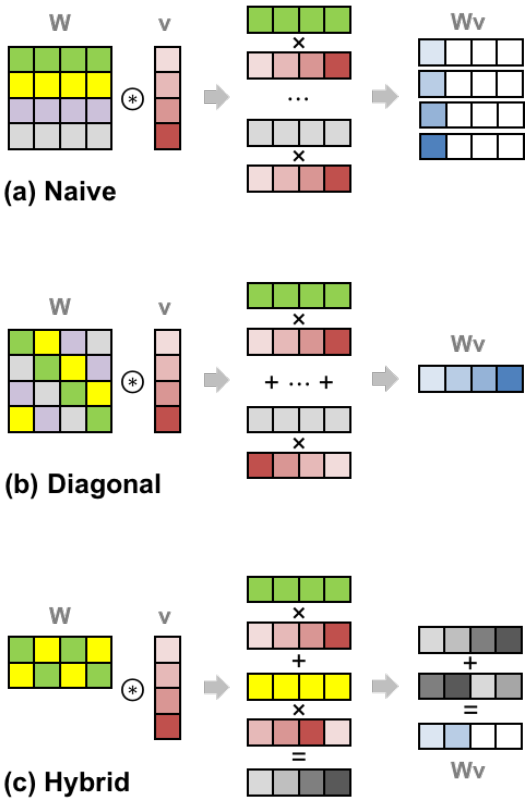

Figure 6: The naïve method is illustrated on the left and the diagonal method of Halevi and Shoup [22] is illustrated on the right. The entries in a single color live in the same ciphertext. The key feature of the diagonal method is that no two elements of the matrix that influence the same output element appear with the same color.

5.2 Input Packing

Before moving on to more complex techniques we describe an orthogonal approach to improve the naïve method when $n_i \ll n$. The idea is to pack multiple copies of the input into a single ciphertext. This allows us better utilization of the slots by computing multiple outputs in parallel.

In detail we can (a) pack n/n_i many different rows into a single plaintext vector; (b) pack n/n_i *copies* of the input vector into a single ciphertext; and (c) perform the rest of the naïve method as-is except that the rotations are not applied to the whole ciphertext but block-by-block (thus requiring $\log(n_i)$ many rotations). Roughly speaking, this achieves communication and computation as if the number of rows of the matrix were $n'_o = (n_o \times n_i)/n$ instead of n_o. When $n_i \ll n$, we have $n'_o \ll n_o$.

The Diagonal Method. The diagonal method as described in the work of Halevi and Shoup [22] (and implemented in [21]) provides another potential solution to the problem of a large number of output ciphertexts. The key high-level idea is to arrange the matrix elements in such a way that after the SIMD scalar multiplications, "interacting elements" of the matrix-vector product never appear in a single ciphertext. Here, "interacting elements" are the numbers that need to be added together to obtain the final result. The rationale is that if this happens, we never need to add two numbers that live in different slots of the same ciphertexts, thus avoiding ciphertext rotation.

To do this, we encode the diagonal of the matrix into a vector which is then SIMD scalar multiplied with the input vector. The second diagonal (namely, the elements $\mathbf{W}_{0,1}, \mathbf{W}_{1,2}, \ldots, \mathbf{W}_{n_o-1,0}$) is encoded into another vector which is then SIMD scalar multiplied with a rotation (by one) of the input vector, and so on. Finally, all these vectors are added together to obtain the output vector *in one shot*.

The cost of the diagonal method is:

- The total cost is n_i SIMD scalar multiplications, $n_i - 1$ rotations (automorphisms), and $n_i - 1$ SIMD additions.
- The noise grows from η to $(\eta + \eta_{\mathsf{rot}}) \cdot \eta_{\mathsf{mult}} \times n_i$ which, for the parameters we use, is larger than that of the naïve method, but much better than the naïve method with output packing. Roughly speaking, the reason is that in the diagonal method, since rotations are performed before scalar multiplication, the noise growth has a $\eta_{\mathsf{rot}} \cdot \eta_{\mathsf{mult}}$ factor whereas in the naïve method, the order is reversed resulting in a $\eta_{\mathsf{mult}} + \eta_{\mathsf{rot}}$ factor.
- Finally, this process produces *a single* ciphertext that has the entire output vector in packed form already.

In our setting (and we believe in most reasonable settings), the additional noise growth is an acceptable compromise given the large gain in the output length and the corresponding gain in the bandwidth and the overall run-time. Furthermore, the fact that all rotations happen on the input ciphertexts prove to be very important for an optimization of [23] we describe in Appendix A, called "hoisting", which lets us amortize the cost of many *input* rotations.

A Hybrid Approach. One issue with the diagonal approach is that the number of Perm is equal to n_i. In the context of FC layers n_o is often much lower than n_i and hence it is desirable to have a method where the Perm is close to n_o. Our hybrid scheme achieves this by combining the best aspects of the naïve and diagonal schemes. We first extended the idea of diagonals for a square matrix to squat rectangular weight matrices as shown in Figure 6 and then pack the weights along these extended diagonals into plaintext vectors. These plaintext vectors are then multiplied with n_o rotations of the input ciphertext similar to the diagonal method. Once this is done we are left with a single ciphertext that contains n/n_o chunks each contains a partial sum of the n_o outputs. We can proceed similar to the naïve method to accumulate these using a "rotate-and-sum" algorithm.

We implement an input packed variant of the hybrid method and the performance and noise growth characteris-

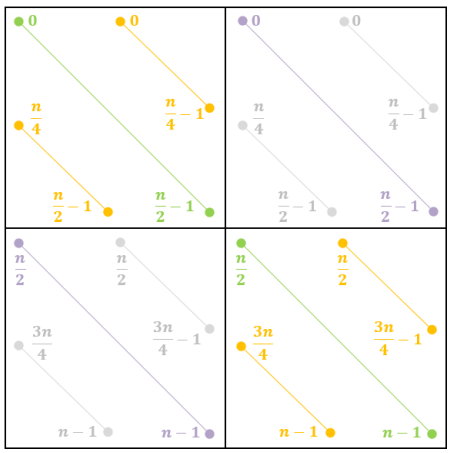

Figure 7: Four example extended digaonals after accounting for the rotation group structure

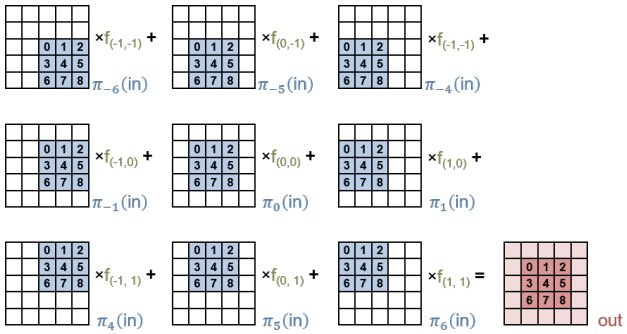

Figure 8: Padded SISO Convolution

tics (following a straightforward derivation) are described in Table 4. We note that hybrid method trades off hoistable input rotations in the Diagonal method for output rotations on distinct ciphertexts (which cannot be "hoisted out"). However, the decrease in the number of input rotations is multiplicative while the corresponding increase in the number of output rotations is the logarithm of the same multiplicative factor. As such, the hybrid method almost always outperforms the Naive and Diagonal methods. We present detailed benchmarks over a selection of matrix sizes in Table 8.

We close this section with two important implementation details. First, recall that in order to enable faster NTT, our parameter selection requires n to be a power of two. As a result the permutation group we have access to is the group of half rotations ($C_{n/2} \times C_2$), i.e. the possible permutations are compositions of rotations by up to $n/2$ for the two $n/2$-sized segments, and swapping the two segments. The packing and diagonal selection in the hybrid approach are modified to account for this by adapting the definition of the extended diagonal to be those entries of $\mathbf{W}$ that would be multiplied by the corresponding entries of the ciphertext when the above Perm operations are performed as shown in Figure 7. Finally, as described in section 3 we control the noise growth in SIMDScMult using plaintext windows for the weight matrix $\mathbf{W}$.

6 Fast Homomorphic Convolutions

We now move on to the implementation of homomorphic kernels for Conv layers. Analogous to the description of FC layers we will start with simpler (and correspondingly less efficient) techniques before moving on to our final optimized implementation. In our setting, the server has access to a plaintext filter and it is then provided encrypted input images, which it must homomorphically convolve with its filter to produce encrypted output images. As a running example for this section we will consider a (f_w, f_h, c_i, c_o)-Conv layer with the *same* padding scheme, where the input is specified by the tuple $(w_i,\ h_i,\ c_i)$. In order to better emphasize the key ideas, we will split our presentation into two parts: first we will describe the single input single output (SISO) case, i.e. $(c_i = 1, c_o = 1)$ followed by the more general case where we have multiple input and output channels, a subset of which may fit within a single ciphertext.

Padded SISO. As seen in section 2, *same* style convolutions require that the input be zero-padded. As such, in this approach, we start with a zero-padded version of the input with $(f_w - 1)/2$ zeros on the left and right edges and $(f_h - 1)/2$ zeros on the top and bottom edges. We assume for now that this padded input image is small enough to fit within a single ciphertext i.e. $(w_i + f_w - 1) \cdot (h_i + f_h - 1) \leq n$ and is mapped to the ciphertext slots in a raster scan fashion. We then compute $f_w \cdot f_h$ rotations of the input and scale them by the corresponding filter coefficient as shown in Figure 8. Since all the rotations are performed on a common input image, they can benefit from the hoisting optimization. Note that similar to the naïve matrix-vector product algorithm, the values on the periphery of the output image leak partial products and must be obscured by adding random values.

Packed SISO. While the above the technique computes the correct 2D-convolution it ends up wasting $(w_i + f_w - 1) \cdot (h_i + f_h - 1) - w_i \cdot h_i$ slots in zero padding. If either the input image is small or if the filter size is large, this can amount to a significant overhead. We resolve this issue by using the ability of our PAHE scheme to multiply different slots with different scalars when performing SIMDScMult. As a result, we can pack the input tightly and generate $f_w \cdot f_h$ rotations. We then multiply these rotated ciphertexts with *punctured plaintexts* which have zeros in the appropriate locations as shown in Figure 9. Accumulating these products gives us a single ciphertext that, as a bonus, contains the convolution result without any leakage of partial information.

Finally, we note that the construction of the punctured

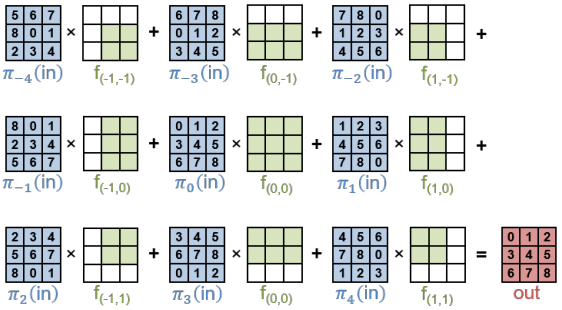

Figure 9: Packed SISO Convolution. (Zeros in the punctured plaintext shown in white.)

Table 3: Comparing SISO 2D-convolutions

	Perm	# slots
Padded	$f_w f_h - 1$	$(w_i + f_w - 1)(h_i + f_h - 1)$
Packed	$f_w f_h - 1$	$w_i h_i$

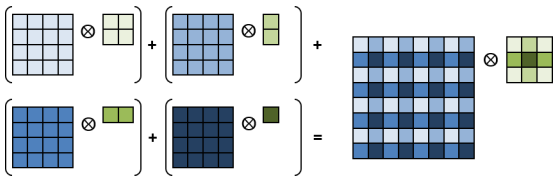

Figure 10: Decomposing a strided convolutions into simple convolutions ($f_w = f_h = 3$ and $s_x = s_y = 2$)

plaintexts does not depend on either the encrypted image or the client key information and as such, the server can precompute these values once for multiple clients. We summarize these results in Table 3.

6.1 Strided Convolutions

We handle strided convolutions by decomposing the strided convolution into a sum of simple convolutions each of which can be handled as above. We illustrate this case for $f_w = f_h = 3$ and $s_x = s_y = 2$ in Figure 10.

6.2 Low-noise Batched Convolutions

We make one final remark on a potential application for padded SISO convolutions. Padded SISO convolutions are computed as a sum of rotated versions of the input images multiplied by corresponding constants $f_{x,y}$. The coefficient domain representation of these plaintext vectors is $(f_{x,y},0,\ldots,0)$. As a result, the noise growth factor is $\eta_{\mathsf{mult}} = f_{x,y} \cdot \sqrt{n}$ as opposed to $p \cdot \sqrt{n}$, consequently noise growth depends only on the value of the filter coefficients and *not* on the size of the plaintext space p. The direct use of this technique precludes the use of channel packing since the filter coefficients are channel dependent. One potential application that can mitigate this issue is when we want to classify a batch of multiple images. In this context, we can pack the same channel from multiple classifications allowing us to use a simple constant filter. This allows us to trade-off classification latency for higher throughput. Note however that similar to padded SISO convolutions, this has two problems: (a) it results in lower slot utilization compare to packed approaches, and (b) the padding scheme reveals the size of the filter.

Now that we have seen how to compute a single 2D-convolution we will look at the more general multi-channel case.

Single Channel per Ciphertext. The straightforward approach for handling the multi-channel case is to encrypt the various channels into distinct ciphertexts. We can then SISO convolve these c_i-ciphertexts with each of the c_o sets of filters to generate c_o output ciphertexts. Note that although we need $c_o \cdot c_i \cdot f_h \cdot f_w$ SIMDAdd and SIMDScMult calls, just $c_i \cdot f_h \cdot f_w$ many Perm operations on the input suffice, since the rotated inputs can be reused to generate each of the c_o outputs. Furthermore, each these rotation can be hoisted and hence we require just c_i many PermDecomp calls and $c_i \cdot f_h \cdot f_w$ many PermAuto calls.

Channel Packing Similar to input-packed matrix-vector products, the computation of multi-channel convolutions can be further sped up by packing multiple channels in a single ciphertext. We represent the number of channels that fit in a single ciphertext by c_n. Channel packing allows us to perform c_n-SISO convolutions in parallel in a SIMD fashion. We maximize this parallelism by using Packed SISO convolutions which enable us to tightly pack the input channels without the need for any additional padding.

For simplicity of presentation, we assume that both c_i and c_o are integral multiples of c_n. Our high level goal is to then start with c_i/c_n input ciphertexts and end up with c_o/c_n output ciphertexts where each of the input and output ciphertexts contains c_n distinct channels. We achieve this in two steps: (a) convolve the input ciphertexts in a SISO fashion to generate $(c_o \cdot c_i)/c_n$ intermediate ciphertexts that contain all the $c_o \cdot c_i$-SISO convolutions and (b) accumulate these intermediate ciphertexts into output ciphertexts.

Since none of the input ciphertexts repeat an input channel, none of the intermediate ciphertexts can contain SISO convolutions corresponding to the same input channel. A similar constraint on the output ciphertexts implies that none of the intermediate ciphertexts contain SISO convolutions corresponding to the same output. In particular, a potential grouping of SISO convolutions that satisfies these constraints is the *diagonal grouping*. More formally the k^{th} intermediate ciphertext in the diagonal grouping contains the following ordered set of c_n-SISO convolutions:

$$\{\, (\lfloor k/c_i \rfloor \cdot c_n + l, \\ \lfloor (k \bmod c_i)/c_n \rfloor \cdot c_n + ((k+l) \bmod c_n)) \mid l \in [0, c_n) \,\}$$

where each tuple (x_o, x_i) represents the SISO convolution

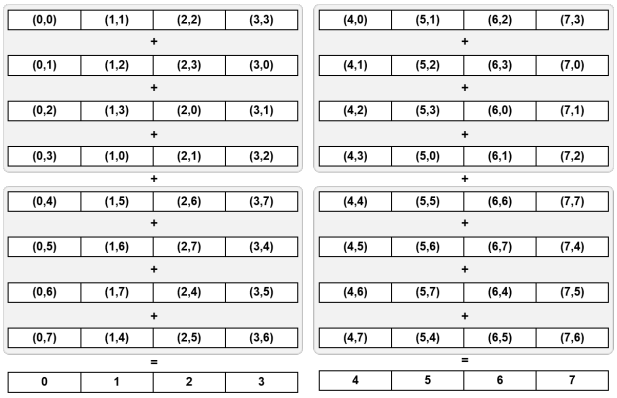

Figure 11: Diagonal Grouping for Intermediate Ciphertexts ($c_i = c_o = 8$ and $c_n = 4$)

corresponding to the output channel x_o and input channel x_i. Given these intermediate ciphertexts, one can generate the output ciphertexts by simply accumulating the c_o/c_n-partitions of c_i consecutive ciphertexts. We illustrate this grouping and accumulation when $c_i = c_o = 8$ and $c_n = 4$ in Figure 11. Note that this grouping is very similar to *the diagonal style of computing matrix vector products*, with single slots now being replaced by entire SISO convolutions.

Since the second step is just a simple accumulation of ciphertexts, the major computational complexity of the convolution arise in the computation of the intermediate ciphertexts. If we partition the set of intermediate ciphertexts into c_n-sized *rotation sets* (shown in grey in Figure 11), we see that each of the intermediate ciphertexts is generated by different rotations of the same input. This observation leads to two natural approaches to compute these intermediate ciphertexts.

Input Rotations. In the first approach, we generate c_n rotations of every input ciphertext and then perform Packed SISO convolutions on each of these rotations to compute all the intermediate rotations required by c_o/c_n rotation sets. Since each of the SISO convolutions requires $f_w \cdot f_h$ rotations, we require a total of $(c_n \cdot f_w \cdot f_h - 1)$ rotations (excluding the trivial rotation by zero) for each of the c_i/c_n inputs. Finally we remark that by using the hoisting optimization we compute all these rotations by performing just c_i/c_n PermDecomp operations.

Output Rotations. The second approach is based on the realization that instead of generating $(c_n \cdot f_w \cdot f_h - 1)$ input rotations, we can reuse $(f_w \cdot f_h - 1)$ rotations in each rotation-set to generate c_n convolutions and then simply rotate $(c_n - 1)$ of these to generate all the intermediate ciphertexts. This approach then reduces the number of input rotations by factor of c_n while requiring $(c_n - 1)$ rotations for each of the $(c_i \cdot c_o)/c_n^2$ rotation sets. Note that while $(f_w \cdot f_h - 1)$ input rotations per input ciphertext can share a common PermDecomp each of the output rotations occur on a distinct ciphertext and cannot benefit from hoisting.

We summarize these numbers in Table 4. The choice between the input and output rotation variants is an interesting trade-off that is governed by the size of the 2D filter. This trade-off is illustrated in more detail with concrete benchmarks in section 7. Finally, we remark that similar to the matrix-vector product computation, the convolution algorithms are also tweaked to work with the half-rotation permutation group and use plaintext windows to control the scalar multiplication noise growth.

7 Implementation and Micro-benchmarks

Next we describe the implementation of the Gazelle framework starting with the chosen cryptographic primitives (7.1). We then describe our evaluation test-bed (7.2) and finally conclude this section with detailed micro-benchmarks (7.3) for all the operations to highlight the individual contributions of the techniques described in the previous sections.

7.1 Cryptographic Primitives

Gazelle needs two main cryptographic primitives for neural network inference: a packed additive homomorphic encryption (PAHE) scheme and a two-party secure computation (2PC) scheme. Parameters for both schemes are selected for a 128-bit security level. For the PAHE scheme we instantiate the Brakerski-Fan-Vercauteren (BFV) scheme [4, 14], with $n = 2048$, 20-bit plaintext modulus, 60-bit ciphertext modulus and $\sigma = 4$ according to the analysis of Section 3.5.

For the 2PC framework, we use Yao's Garbled circuits [44]. The main reason for choosing Yao over Boolean secret sharing schemes (such as the Goldreich-Micali-Wigderson protocol [19] and its derivatives) is that the constant number of rounds results in good performance over long latency links. Our garbling scheme is an extension of the one presented in JustGarble [3] which we modify to also incorporate the Half-Gates optimization [45]. We base our oblivious transfer (OT) implementation on the classic Ishai-Kilian-Nissim-Petrank (IKNP) [27] protocol from libOTe [33]. Since we use 2PC for implementing the ReLU, MaxPool and FHE-2PC transformation gadget, our circuit garbling phase only depends on the neural network topology and is independent of the client input. As such, we move it to the offline phase of the computation while the OT Extension and circuit evaluation is run during the online phase of the computation.

7.2 Evaluation Setup

All benchmarks were generated using c4.xlarge AWS instances which provide a 4-threaded execution environment (on an Intel Xeon E5-2666 v3 2.90GHz CPU) with 7.5GB of system memory. Our experiments were conducted using Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1041-aws)

Table 4: Comparing multi-channel 2D-convolutions

	PermDecomp	Perm	#in_ct	#out_ct
One Channel per CT	c_i	$(f_w f_h - 1) \cdot c_i$	c_i	c_o
Input Rotations	$\frac{c_i}{c_n}$	$(c_n f_w f_h - 1) \cdot \frac{c_i}{c_n}$	$\frac{c_i}{c_n}$	$\frac{c_o}{c_n}$
Output Rotations	$\left(1 + \frac{(c_n-1)\cdot c_o}{c_n}\right)\frac{c_i}{c_n}$	$\left(f_w f_h - 1 + \frac{(c_n-1)\cdot c_o}{c_n}\right)\frac{c_i}{c_n}$	$\frac{c_i}{c_n}$	$\frac{c_o}{c_n}$

Table 5: Fast Reduction for NTT and Inv. NTT

Operation	Fast Reduction		Naive Reduction		Speedup
	t (μs)	cyc/bfly	t (μs)	cyc/bfly	
NTT (q)	57	14.68	393	101.18	6.9
Inv. NTT (q)	54	13.90	388	99.89	7.2
NTT (p)	43	11.07	240	61.79	5.6
Inv. NTT (p)	38	9.78	194	49.95	5.1

Table 6: FHE Microbenchmarks

Operation	Fast Reduction		Naive Reduction		Speedup
	t (μs)	cyc/slot	t (μs)	cyc/slot	
KeyGen	232	328.5	952	1348.1	4.1
Encrypt	186	263.4	621	879.4	3.3
Decrypt	125	177.0	513	726.4	4.1
SIMDAdd	5	8.1	393	49.7	6.1
SIMDScMult	10	14.7	388	167.1	11.3
PermKeyGen	466	659.9	1814	2568.7	3.9
Perm	268	379.5	1740	2463.9	6.5
PermDecomp	231	327.1	1595	2258.5	6.9
PermAuto	35	49.6	141	199.7	4.0

and our library was compiled using GCC 5.4.0 using the '-O3' optimization setting and enabling support for the AES-NI instruction set. Our schemes are evaluated in the LAN setting similar to previous work with both instances in the us-east-1a availability zone.

7.3 Micro-benchmarks

In order to isolate the impact of the various techniques and identify potential optimization opportunities, we first present micro-benchmarks for the individual operations.

Arithmetic and PAHE Benchmarks. We first benchmark the impact of the faster modular arithmetic on the NTT and the homomorphic evaluation run-times. Table 5 shows that the use of a pseudo-Mersenne ciphertext modulus coupled with lazy modular reduction improves the NTT and inverse NTT by roughly 7×. Similarly Barrett reduction for the plaintext modulus improves the plaintext NTT runtimes by more than 5×. These run-time improvements are also reflected in the performance of the primitive homomorphic operations as shown in Table 6.

Table 7 demonstrates the noise performance trade-off inherent in the permutation operation. Note that an individual permutation after the initial decomposition is roughly 8-9× faster than a permutation without any pre-computation. Finally we observe a linear growth in the run-time of the permutation operation with an increase in the number of windows, allowing us to trade off noise performance for run-time if few future operations are desired on the permuted ciphertext.

Table 7: Permutation Microbenchmarks

# windows	PermKeyGen	Key Size	PermAuto	Noise
	t (μs)	kB	t (μs)	bits
3	466	49.15	35	29.3
6	925	98.30	57	19.3
12	1849	196.61	100	14.8

Linear Algebra Benchmarks. Next we present micro-benchmarks for the linear algebra kernels. In particular we focus on matrix-vector products and 2D convolutions since these are the operations most frequently used in neural network inference. Before performing these operations, the server must perform a one-time *client-independent setup* that pre-processes the matrix and filter coefficients. In contrast with the offline phase of 2PC, this computation is NOT repeated per classification or per client and can be performed without any knowledge of the client keys. In the following results, we represent the time spent in this amortizable setup operation as t_{setup}. Note that $t_{offline}$ for both these protocols is zero.

The matrix-vector product that we are interested in corresponds to the multiplication of a plaintext matrix with a packed ciphertext vector. We first start with a comparison of three matrix-vector multiplication techniques:

1. **Naive**: Every slot of the output is generated independently by computing an inner-product of a row of the matrix with ciphertext column vector.
2. **Diagonal**: Rotations of the input are multiplied by the generalized diagonals from the plaintext matrix and added to generate a packed output.
3. **Hybrid**: Use the diagonal approach to generate a single output ciphertext with copies of the output partial sums. Use the naive approach to generate the final output from this single ciphertext

Table 8: Matrix Multiplication Microbenchmarks

		#in_rot	#out_rot	#mac	t_{online}	t_{setup}
2048×1	**N**	0	11	1	7.9	16.1
	D	2047	0	2048	383.3	3326.8
	H	0	11	1	8.0	16.2
1024×128	**N**	0	1280	128	880.0	1849.2
	D	1023	1024	2048	192.4	1662.8
	H	63	4	64	16.2	108.5
1024×16	**N**	0	160	16	110.3	231.4
	D	1023	1024	2048	192.4	1662.8
	H	7	7	8	7.8	21.8
128×16	**N**	0	112	16	77.4	162.5
	D	127	128	2048	25.4	206.8
	H	0	7	1	5.3	10.5

Table 9: Convolution Microbenchmarks

Input (W×H, C)	Filter (W×H, C)	Algorithm	t_{online} (ms)	t_{setup} (ms)
(28×28,1)	(5×5,5)	**I**	14.4	11.7
		O	9.2	11.4
(16×16,128)	(1×1,128)	**I**	107	334
		O	110	226
(32×32,32)	(3×3,32)	**I**	208	704
		O	195	704
(16×16,128)	(3×3,128)	**I**	767	3202
		O	704	3312

We compare these techniques for the following matrix sizes: 2048×1, 1024×128, 128×16. For all these methods we report the online computation time and the time required to setup the scheme in milliseconds. Note that this setup needs to be done exactly once per network and need not be repeated per inference. The naive scheme uses a 20bit plaintext window (w_{pt}) while the diagonal and hybrid schemes use 10bit plaintext windows. All schemes use a 7bit relinearization window (w_{relin}).

Finally we remark that our matrix multiplication scheme is extremely parsimonious in the online bandwidth. The two-way online message sizes for all the matrices are given by $(w+1) * ct_{sz}$ where ct_{sz} is the size of a single ciphertext (32 kB for our parameters).

Next we compare the two techniques we presented for 2D convolution: input rotation **(I)** and output rotation **(O)** in Table 9. We present results for four convolution sizes with increasing complexity. Note that the 5×5 convolution is strided convolution with a stride of 2. All results are presented with a 10bit w_{pt} and a 8bit w_{relin}.

As seen from Table 9, the output rotation variant is usually the faster variant since it reuses the same input multiple times. Larger filter sizes allow us to save more rotations and hence experience a higher speed-up, while for the 1×1 case the input rotation variant is faster. Finally, we note that in all cases we pack both the input and output activations using the minimal number of ciphertexts.

Table 10: Activation and Pooling Microbenchmarks

Algorithm	Outputs	$t_{offline}$ (ms)	t_{online} (ms)	$BW_{offline}$ (MB)	BW_{online} (MB)
Square	2048	0.5	1.4	0	0.093
ReLU	1000	89	15	5.43	1.68
	10000	551	136	54.3	16.8
MaxPool	1000	164	58	15.6	8.39
	10000	1413	513	156.0	83.9

Square, ReLU and MaxPool Benchmarks. We round our discussion of the operation micro-benchmarks with the various activation functions we consider. In the networks of interest, we come across two major activation functions: Square and ReLU. Additionally we also benchmark the MaxPool layer with (2×2)-sized windows.

For square pooling, we implement a simple interactive protocol using our additively homomorphic encryption scheme. For ReLU and MaxPool, we implement a garbled circuit based interactive protocol. The results for both are presented in Table 10.

8 Network Benchmarks and Comparison

Next we compose the individual layers from the previous sections and evaluate complete networks. For ease of comparison with previous approaches, we report runtimes and network bandwidth for MNIST and CIFAR-10 image classification tasks. We segment our comparison based on the CNN topology. This allows us to clearly demonstrate the speedup achieved by Gazelle as opposed to gains through network redesign.

The MNIST Dataset. MNIST is a basic image classification task where we are provided with a set of 28×28 grayscale images of handwritten digits in the range $[0-9]$. Given an input image our goal is to predict the correct handwritten digit it represents. We evaluate this task using four published network topologies which use a combination of FC and Conv layers:

A 3-FC with square activation from [30].
B 1-Conv and 2-FC with square activation from [18].
C 1-Conv and 2-FC with ReLU activation from [36].
D 2-Conv and 2-FC with ReLU and MaxPool from [29].

Runtime and the communication required for classifying a single image for these four networks are presented in table 11.

For all four networks we use a 10bit w_{pt} and a 9bit w_{relin}.

Table 11: MNIST Benchmark

	Framework	Runtime (s)			Communication (MB)		
		Offline	Online	Total	Offline	Online	Total
A	SecureML	4.7	0.18	4.88	-	-	-
	MiniONN	0.9	0.14	1.04	3.8	12	47.6
	Gazelle	0	0.03	0.03	0	0.5	0.5
B	CryptoNets	-	-	297.5	-	-	372.2
	MiniONN	0.88	0.4	1.28	3.6	44	15.8
	Gazelle	0	0.03	0.03	0	0.5	0.5
C	DeepSecure	-	-	9.67	-	-	791
	Chameleon	1.34	1.36	2.7	7.8	5.1	12.9
	Gazelle	0.15	0.05	0.20	5.9	2.1	8.0
D	MiniONN	3.58	5.74	9.32	20.9	636.6	657.5
	ExPC	-	-	5.1	-	-	501
	Gazelle	0.481	0.33	0.81	47.5	22.5	70.0

Table 12: CIFAR-10 Benchmark

	Framework	Runtime (s)			Communication (MB)		
		Offline	Online	Total	Offline	Online	Total
A	MiniONN	472	72	544	3046	6226	9272
	Gazelle	9.34	3.56	12.9	940	296	1236

Networks A and B use only the square activation function allowing us to use a much simpler AHE base interactive protocol, thus avoiding any use of GC's. As such we only need to transmit short ciphertexts in the online phase. Similarly our use of the AHE based FC and Conv layers as opposed to multiplications triples results in 5-6× lower latency compared to [29] and [30] for network A. The comparison with [18] is even more the stark. The use of AHE with interaction acting as an implicit bootstraping stage allows for aggressive parameter selection for the lattice based scheme. This results in over 3 orders of magnitude savings in both the latency and the network bandwidth.

Networks C and D use ReLU and MaxPool functions which we implement using GC. However even for these the network our efficient FC and Conv implementation allows us roughly 30× and 17× lower runtime when compared with [32] and [29] respectively. Furthermore we note that unlike [32] our solution does not rely on a trusted third party.

The CIFAR-10 Dataset. The CIFAR-10 task is a second commonly used image classification benchmark that is substantially more complicated than the MNIST classification task. The task consists of classifying 32×32 color with 3 color channels into 10 classes such as automobiles, birds, cats, etc. For this task we replicate the network topology from [29] to offer a fair comparison. We use a 10bit w_{pt} and a 8bit w_{relin}.

We note that the complexity of this network when measured by the number of multiplications is 500× that used in the MNIST network from [36], [32]. By avoiding the need for multiplication triples Gazelle offers a 50× faster offline phase and a 20× lower latency per inference showing that our results from the smaller MNIST networks scale to larger networks.

9 Conclusions and Future Work

In conclusion, this work presents Gazelle, a low-latency framework for secure neural network inference. Gazelle uses a judicious combination of packed additively homomorphic encryption (PAHE) and garbled circuit based two-party computation (2PC) to obtain $20-30\times$ lower latency and $2.5-88\times$ lower online bandwidth when compared with multiple recent 2PC-based state-of-art secure network inference solutions [29, 30, 32, 36], and more than 3 orders of magnitude lower latency and 2 orders of magnitude lower bandwidth than purely homomorphic approaches [18]. We briefly recap the key contributions of our work that enable this improved performance:

1. Selection of prime moduli that simultaneously allow SIMD operations, low noise growth and division-free and lazy modular reduction.
2. Avoidance of ciphertext-ciphertext multiplications to reduce noise growth.
3. Use of secret-sharing and interaction to emulate a lightweight bootstrapping procedure allowing us to evaluate deep networks composed of many layers.
4. Homomorphic linear algebra kernels that make efficient use of the automorphism structure enabled by a power-of-two slot-size.
5. Sparing use of garbled circuits limited to ReLU and MaxPool functions with linear-size Boolean circuits.
6. A compact garbled circuit-based transformation gadget that allows us to securely compose the PAHE-based and garbled circuit based layers.

There are a large number of natural avenues to build on our work including handling neural networks with larger input sizes and building a framework to automatically compile neural networks into secure inference protocols.

Acknowledgments

We thank Kurt Rohloff, Yuriy Polyakov and the PALISADE team for providing us with access to the PALISADE library. We thank Shafi Goldwasser, Rina Shainski and Alon Kaufman for delightful discussions. We thank our sponsors, the Qualcomm Innovation Fellowship and Delta Electronics for supporting this work.

References

[1] Albrecht, M. R., Player, R., and Scott, S. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology 9*, 3 (2015), 169–203.

[2] Angelini, E., di Tollo, G., and Roli, A. A neural network approach for credit risk evaluation. *The Quarterly Review of Economics and Finance 48*, 4 (2008), 733 – 755.

[3] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher.

In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 478–492.

[4] BRAKERSKI, Z. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (2012), pp. 868–886.

[5] BRAKERSKI, Z., GENTRY, C., AND VAIKUNTANATHAN, V. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS* (2012).

[6] BRAKERSKI, Z., AND VAIKUNTANATHAN, V. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS* (2011).

[7] CHILLOTTI, I., GAMA, N., GEORGIEVA, M., AND IZABACHÈNE, M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I* (2016), pp. 3–33.

[8] CHILLOTTI, I., GAMA, N., GEORGIEVA, M., AND IZABACHENE, M. Tfhe: Fast fully homomorphic encryption over the torus, 2017. https://tfhe.github.io/tfhe/.

[9] DAMGARD, I., PASTRO, V., SMART, N., AND ZACHARIAS, S. The spdz and mascot secure computation protocols, 2016. https://github.com/bristolcrypto/SPDZ-2.

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

[11] DUCAS, L., AND STEHLÉ, D. Sanitization of FHE ciphertexts. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I* (2016), pp. 294–310.

[12] EJGENBERG, Y., FARBSTEIN, M., LEVY, M., AND LINDELL, Y. Scapi: Secure computation api, 2014. https://github.com/cryptobiu/scapi.

[13] ESTEVA, A., KUPREL, B., NOVOA, R. A., KO, J., SWETTER, S. M., BLAU, H. M., AND THRUN, S. Dermatologist-level classification of skin cancer with deep neural networks. *Nature 542*, 7639 (2017), 115–118.

[14] FAN, J., AND VERCAUTEREN, F. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive 2012* (2012), 144.

[15] GENTRY, C. A fully homomorphic encryption scheme. PhD Thesis, Stanford University, 2009.

[16] GENTRY, C., HALEVI, S., AND SMART, N. P. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings* (2012), pp. 465–482.

[17] GENTRY, C., HALEVI, S., AND VAIKUNTANATHAN, V. A simple BGN-type cryptosystem from LWE. In *EUROCRYPT* (2010).

[18] GILAD-BACHRACH, R., DOWLIN, N., LAINE, K., LAUTER, K. E., NAEHRIG, M., AND WERNSING, J. 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* (2016), pp. 201–210.

[19] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC* (1987).

[20] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof systems. *SIAM J. Comput. 18*, 1 (1989), 186–208.

[21] HALEVI, S., AND SHOUP, V. An implementation of homomorphic encryption, 2013. https://github.com/shaih/HElib.

[22] HALEVI, S., AND SHOUP, V. Algorithms in HElib. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I* (2014), pp. 554–571.

[23] HALEVI, S., AND SHOUP, V., 2017. Presentation at the Homomorphic Encryption Standardization Workshop, Redmond, WA, July 2017.

[24] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[25] HENECKA, W., SADEGHI, A.-R., SCHNEIDER, T., WEHRENBERG, I., ET AL. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 451–462.

[26] INDYK, P., AND WOODRUFF, D. P. Polylogarithmic private approximations and efficient matching. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings* (2006), pp. 245–264.

[27] ISHAI, Y., KILIAN, J., NISSIM, K., AND PETRANK, E. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings* (2003), pp. 145–161.

[28] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.* (2012), pp. 1106–1114.

[29] LIU, J., JUUTI, M., LU, Y., AND ASOKAN, N. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), pp. 619–631.

[30] MOHASSEL, P., AND ZHANG, Y. Secureml: A system for scalable privacy-preserving machine learning. In *2017*

IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017 (2017), pp. 19–38.

[31] PAILLIER, P. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology – EUROCRYPT '99* (1999), pp. 223–238.

[32] RIAZI, M. S., WEINERT, C., TKACHENKO, O., SONGHORI, E. M., SCHNEIDER, T., AND KOUSHANFAR, F. Chameleon: A hybrid secure computation framework for machine learning applications. Cryptology ePrint Archive, Report 2017/1164, 2017. https://eprint.iacr.org/2017/1164.

[33] RINDAL, P. Fast and portable oblivious transfer extension, 2016. https://github.com/osu-crypto/libOTe.

[34] RIVEST, R. L., ADLEMAN, L., AND DERTOUZOS, M. L. On data banks and privacy homomorphisms. *Foundations of Secure Computation* (1978).

[35] ROHLOFF, K., AND POLYAKOV, Y. *The PALISADE Lattice Cryptography Library*, 1.0 ed., 2017. Library available at https://git.njit.edu/palisade/PALISADE.

[36] ROUHANI, B. D., RIAZI, M. S., AND KOUSHANFAR, F. Deepsecure: Scalable provably-secure deep learning. *CoRR abs/1705.08963* (2017).

[37] SADEGHI, A., SCHNEIDER, T., AND WEHRENBERG, I. Efficient privacy-preserving face recognition. In *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers* (2009), pp. 229–244.

[38] SCHROFF, F., KALENICHENKO, D., AND PHILBIN, J. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015* (2015), pp. 815–823.

[39] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[40] SZE, V., CHEN, Y., YANG, T., AND EMER, J. S. Efficient processing of deep neural networks: A tutorial and survey. *CoRR abs/1703.09039* (2017).

[41] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., AND RABINOVICH, A. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)* (2015).

[42] TRAMÈR, F., ZHANG, F., JUELS, A., REITER, M. K., AND RISTENPART, T. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 601–618.

[43] V, G., L, P., M, C., AND ET AL. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA 316*, 22 (2016), 2402–2410.

[44] YAO, A. C. How to generate and exchange secrets (extended abstract). In *FOCS* (1986).

[45] ZAHUR, S., ROSULEK, M., AND EVANS, D. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II* (2015), pp. 220–250.

A The Halevi-Shoup Hoisting Optimization

The hoisting optimization reduces the cost of the ciphertext rotation when the *same* ciphertext must be rotated by multiple shift amounts. The idea, roughly speaking, is to "look inside" the ciphertext rotation operation, and hoist out the part of the computation that would be common to these rotations and then compute it only once thus amortizing it over many rotations. It turns out that this common computation involves computing the NTT^{-1} (taking the ciphertext to the coefficient domain), followed by a $\mathsf{w}_{\mathsf{relin}}$-bit decomposition that splits the ciphertext $\lceil(\log_2 q)/\mathsf{w}_{\mathsf{relin}}\rceil$ ciphertexts and finally takes these ciphertexts back to the evaluation domain using separate applications of NTT. The parameter $\mathsf{w}_{\mathsf{relin}}$ is called the relinearization window and represents a tradeoff between the speed and noise growth of the Perm operation. This computation, which we denote as $\mathsf{PermDecomp}$, has $\Theta(n\log n)$ complexity because of the number theoretic transforms. In contrast, the independent computation in each rotation, denoted by $\mathsf{PermAuto}$, is a simple $\Theta(n)$ multiply and accumulate operation. As such, hoisting can provide substantial savings in contrast with direct applications of the Perm operation and this is also borne out by the benchmarks in Table 7.

B Circuit Privacy

We next provide some details on our light-weight circuit privacy solution. At a high level BFV ciphertexts look like a tuple of ring elements (a,b) where a is chosen uniformly at random and b encapsulates the plaintext and the ciphertext noise. Both a and the ciphertext noise are modified in a circuit dependent fashion during the process of homomorphic computation and thus may violate circuit privacy. We address the former by simply adding a fresh public-key encryption of zero to the ciphertext to re-randomize a. Information leakage through the noise is handled through interactive decryption. The BFV decryption circuit is given by $\lceil(a\cdot s+b)/\Delta\rfloor$ where s is the secret key and $\Delta=\lfloor(p/q)\rfloor$. Our approach splits the interactive computation of this circuit into 2 phases. First we send the re-randomized a back to the client who multiplies it with s to $a\cdot s$. We then use a garbled circuit to add this to b. We leverage the fact that Δ is public to avoid an expensive division inside the garbled circuit. In particular both parties can compute the quotients and remainders modulo Δ of their respective inputs and then interactively evaluate a garbled circuit whose size is $\Omega(n\cdot q)$. Note that in contrast the naive decryption circuit is $\Omega(n^2\cdot q)$ sized even without accounting for the division by Δ.