The AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs

Ahmad Al Badawi [1], Jin Chao [1], Jie Lin [1], Chan Fook Mun [1], Jun Jie Sim [1], Benjamin Hong Meng Tan [1], Xiao Nan [1], Khin Mi Mi Aung* [1], and Vijay Ramaseshan Chandrasekhar* [1]

{Ahmad_Al_Badawi, Jin_Chao, lin-j, Chan_Fook_Mun, Sim_Jun_Jie, Benjamin_Tan, Mi_Mi_Aung, vijay}@i2r.a-star.edu.sg

[1]Institute for Infocomm Research (I^{2}R), A*STAR, Singapore

February 4, 2019

Abstract

Machine/Deep Learning (M/DL) as a Service stands as a promising solution for cloud-based inference applications. In this setting, the cloud has a pre-learned model and large computing capacity whereas the user has the samples on which she wants to run the model. The main concern with these systems is related to the privacy of the input data. Our work offers secure, efficient and non-interactive privacy-preserving solution by employing high-end technologies such as modern cryptographic primitives, advanced DL techniques and high-performance computing hardware. Namely, we use Fully Homomorphic Encryption (FHE), Convolutional Neural Networks (CNNs) and Graphics Processing Units (GPUs).
FHE, with its widely-known feature of non-interactively computing on encrypted data, empowers a wide range of privacy-concerned cloud applications including M/DLaaS. This comes at a high cost since FHE includes highly-intensive computation that requires enormous computing power. Although the literature includes a number of proposals to run CNNs on encrypted data, the performance is still far from satisfactory. In this paper, we push the level up and show how to accelerate the performance of running CNNs on encrypted data using GPUs. We evaluated two CNNs to classify homomorphically the MNIST and CIFAR-10 datasets into 10 classes. We used a number of techniques such as low-precision training, unified training and testing network, optimized FHE parameters and a very efficient GPU implementation to achieve high performance. Our solution achieved high security level (> 80 bit) and reasonable classification accuracy (99%) and (77.55%) for MNIST and CIFAR-10, respectively. In terms of performance, our best results show that we could classify the entire testing datasets in 6.46 seconds (resp. 3044 seconds), with per-image amortized time (0.788 milliseconds) (resp. 371 milliseconds) for MNIST and CIFAR-10, respectively.

Fully Homomorphic Encryption, Deep Learning, Encrypted Convolutional Neural Networks, Privacy-preserving Computing, GPU Acceleration

1 Introduction

Machine/Deep Learning (M/DL) has empowered a wide range of applications from labelling systems, web searches, content filtering to recommendation systems on entertainment and ecommerce systems. The prominent feature of this technology is its ability to perform tasks that we humans can do seamlessly such as labelling, identification and recommendation. In order to do that, this technology relies on complex models that are able to capture specific features from input data. Building these models is a complex task that requires substantial domain expertise of several disciplines starting from neurology to computer science. Therefore, there is an evolving trend toward leasing instead of building these models, which made Machine/Deep Learning as a Service (M/DLaaS) an indispensable solution.

The cloud stands as a good platform to either create or host pre-learned models as it offers cheap data storage, near-zero deployment cost and high computational services [23]. However, it has some drawbacks and raises important questions that need to be resolved. One of the main questions is that cloud platforms do not guarantee data privacy. In the DLaaS setting, the user uploads her data to the cloud which in turn evaluates the model on the input data and sends the results back to the user. At every step along this process, there are numerous opportunities for attackers and other malicious actors to compromise the data.

As a motivating example, suppose there is a financial service provider who claims to invalidate the efficient market hypothesis and owns a model that can be used to predict the market value of a designated class of companies with high accuracy rate. The service provider would ideally want to monetize their service by hosting it on the cloud and providing paid inferences according to certain service agreements with the clients. Moreover, a potential client may feel reluctant to use the service and share her confidential data with the cloud despite the potential return value. An ideal solution to this problem shall protect both the model and private data. Moreover, it has to reveal nothing about the prediction result. Besides, the solution is preferred to be non-interactive, i.e., does not require the client to be involved in the computation except for providing input and receiving output. Finally, the solution has to be efficient with reasonable response time.

The main objective of our work is to provide secure, efficient and non-interactive solution to the aforementioned scenario. Our solution reveals nothing about the input, say x, to the cloud nor it does about the model, say f, to the client except what can be learned from input x and $f(x)$. The system is also secure against passive adversaries sniffing the communication between the cloud and the client. However, we assume that the cloud is semi-honest, i.e., it follows the protocol with the exception that it tries to learn the private inputs from clients. Our solution is also efficient and reasonably practical especially if multiple predictions are to be made simultaneously. Lastly, it is non-interactive since the client needs only to interact with the cloud to provide the input data and receive the output.

There are at least three possible methods to tackle the aforementioned problem: 1) Trusted Computing Base (TCB) [36], such as INTEL Software Guard Extensions (SGX) or AMD Secure Encrypted Virtualization (SEV), which provides hardware-based primitives that can be used to execute private data and/or code in shielded environments, called enclaves, 2) Multi-Party Computation (MPC) protocols which are algorithmic solutions that employ cryptography to jointly evaluate a certain function between multiple parties without revealing any party's private input and 3) Fully Homomorphic Encryption (FHE) that allows anyone to evaluate a certain function on encrypted data without decryption. TCB-based solutions are very efficient and suitable for the cloud computing paradigm. However, several attacks have been recently found to affect TCB by allowing unprivileged programs to extract memory content that is only accessible to privileged programs [27, 32, 13] which raises more concerns about this technology. MPC solutions are interactive solutions that require a fair amount of communication between the participating parties, i.e., they are communication-bound. On the other hand, FHE-based solutions are similar to MPC solutions except that they are non-interactive. However, they are considered compute-bound as they require an enormous amount of computation. As computation is typically simpler to tackle than communication, this work employs a combination of technologies to design a FHE-based solution that is secure, efficient and non-interactive.

FHE-based Privacy-preserving machine learning was considered previously by Graepel *et al.* [19] and Aslett *et al.* [6]. Following them, Dowlin *et al.* [16] proposed CryptoNets, the first neural network over encrypted data, providing a method to do the inference phase of privacy-preserving deep learning. Since then, others [33, 10, 25, 22, 24] have applied a variety of cryptographic techniques, such as secure MPC and oblivious transfers, to achieve similar goals. The main drawback of these FHE-based solutions is the computational overhead. For instance, CryptoNets required 570 seconds to evaluate a FHE-friendly model on encrypted samples from the MNIST dataset [30] at security level (80-bit). In addition, their network required a high plaintext modulus ($t \approx 2^{80}$) which had to be decomposed via the Chinese Remainder Theorem (CRT) into two smaller (2^{40}) moduli. Therefore, they had to run the network twice to evaluate the model. Moreover, the scheme they used (YASHE$'$) is not recommended to use due to recent lattice subfield attacks [4]. To this end, we design a solution that overcomes the aforementioned problems. For instance, our solution is more efficient and requires only 6.46 seconds to evaluate the MNIST model with security level > 91 bits. Also, it requires a much smaller plaintext modulus ($t \approx 2^{43}$) that can be used directly without CRT decomposition.

Just as AlexNet by Krizhevsky *et al.* [29] showed how image classification is viable by running Convolutional Neural Networks (CNN) on GPUs, our main contribution in this work is to show that privacy-preserving deep learning is not only possible on GPUs but can also be dramatically accelerated and offers a way towards efficient DLaaS. We follow the framework put forward in CryptoNets [16] and apply our GPU-accelerated FHE techniques to realize efficient Homomorphic Convolutional Neural Networks (HCNNs).

Although the framework is available, there are still a number of challenges to be addressed to realize practical HCNNs. FHE, first realized by Gentry [18] almost 10 years ago, allows arbitrary computation on encrypted data. Informally, it works as follows. Encryption masks the input data, called a plaintext, by a random error sampled from some distribution, resulting in a ciphertext that reveals nothing about what it encrypts. Decryption uses the secret key to filter out the noise and retrieve the plaintext as long as the noise is within a certain bound. Note that during computation, the noise in ciphertexts grows but in a controlled manner. At some point, it grows to a point where no further computation can be done without resulting in a decryption failure. Bootstrapping can be used to refresh a ciphertext with large noise into one with less noise that can be used for further computation. By doing this indefinitely, theoretically, any function can be computed on encrypted data.

As bootstrapping is extremely expensive, this approach is still impractical and bootstrapping is usually avoided in most practical settings. Instead, the class of functions that can be evaluated is restricted to depth L arithmetic circuits, yielding a levelled FHE scheme that can be parameterized to support circuits of depth $leqL$. For performance, L should be minimized which means that we have to carefully design HCNNs with this in mind. Furthermore, the model of computation in FHE, arithmetic circuits with homomorphic addition (HAdd) and multiplication (HMult) gates, is not compatible with non-polynomial functions such as sigmoid, ReLU and max. This means that we should use polynomial approximations to the activation functions where possible.

Besides that, we have to encode decimals in a form that is compatible with FHE plaintext data, which are usually integers. These can have high precision which means that they will require integers of large bit-size to represent them in the commonly used *scalar encoding*. In this encoding, decimals are transformed into integers by multiplying them with a scaling factor Δ and then operated on with HAdd and HMult normally. The main drawback of this encoding is that we cannot re-scale encoded data mid-computation; therefore, successive homomorphic operations will cause data size to increase rapidly. Managing this scaling expansion is a necessary step towards scaling HCNNs to larger datasets and deeper neural networks.

Our Contributions.

1. We present the first GPU-accelerated Homomorphic Convolutional Neural Networks (HCNN) that runs a pre-learned model on encrypted data from the MNIST and CIFAR-10 datasets.
2. We provide a rich set of optimization techniques to enable easy designs of HCNN and reduce the overall computational overhead. These include low-precision training, optimized choice of FHE scheme and parameters, and a GPU-accelerated implementation.
3. We reduced the HCNN for the MNIST dataset to only 5 layers deep for both training and inference, smaller than CryptoNets [16] which used 9 layers during training. For CIFAR-10, we provide an 11-layer FHE-friendly network which can be used for both training and inference as well.
4. Our implementations shows high performance in terms of runtime and accuracy. On the MNIST dataset, our HCNN can evaluate the entire dataset in 8.10 seconds, 3.53× faster than E2DM [24]. On CIFAR-10 dataset, our HCNN requires 3044 seconds, 3.83× faster than CryptoDL [21].

Related Work. The research in the area of privacy-preserving deep learning can be roughly divided into two camps: those using homomorphic encryption or combining it with secure multi-party computation (MPC) techniques. Most closely related to our work are CryptoNets by Dowlin *et al.* [16], FHE-DiNN by Bourse *et al.* [10] and E2DM by Jiang *et al.* [24], who focus on using only fully homomorphic encryption to address this problem. Dowlin *et al.* [16] were the first to propose using FHE to achieve privacy-preserving deep learning, offering a framework to design neural networks that can be run on encrypted data. They proposed using polynomial approximations of the most widespread ReLU activation function and using pooling layers only during the training phase to reduce the circuit depth of their neural network. However, they used the YASHE′ scheme by Bos *et al.* [9] which is no longer secure due to attacks proposed by Albrecht *et al.* [4]. Also, they require a large plaintext modulus of over 80 bits to accommodate the output result of their neural network's. This makes it very difficult to scale to deeper networks since intermediate layers in those networks will quickly reach several hundred bits with such settings.

Following them, Bourse *et al.* [10] proposed a new type of neural network called discretized neural networks (DiNN) for inference over encrypted data. Weights and inputs of traditional CNNs are discretized into elements in $\{-1, 1\}$ and the fast bootstrapping of the TFHE scheme proposed by Chilotti *et al.* [14] was exploited to double as an activation function for neurons. Each neuron computes a weighted sum of its inputs and the activation function is the sign function, $\mathsf{sign}(z)$ which outputs the sign of the input z. Although this method can be applied to arbitrarily deep networks, it suffers from lower accuracy, achieving only 96.35% accuracy on the MNIST dataset with lower amortized performance. Very recently, Jiang *et al.* [24] proposed a new method for matrix multiplication with FHE and evaluated a neural network on the MNIST data set using this technique. They also considered packing an entire image into a single ciphertext compared to the approach of Dowlin *et al.* [16] who put only one pixel per ciphertext but evaluated large batches of images at a time. They achieved good performance, evaluating 64 images in slightly under 29 seconds but with worse amortized performance.

Some of the main limitations of pure FHE-based solutions is the need to approximate non-polynomial activation functions and high computation time. Addressing these problems, Liu *et al.* [33] proposed MiniONN, a paradigm shift in securely evaluating neural networks. They take commonly used protocols in deep learning and transform them into oblivious protocols. With MPC, they could evaluate neural networks without changing the training phase, preserving accuracy since there is no approximation needed for activation functions. However, MPC comes with its own set of drawbacks. In this setting, each computation requires communication between the data owner and model owner, thus resulting in high bandwidth usage. In a similar vein, Juvekar *et al.* [25] designed GAZELLE. Instead of applying levelled FHE, they alternate between an additive homomorphic encryption scheme for convolution-type layers and garbled circuits for activation and pooling layers. This way, communication complexity is reduced compared to MiniONN but unfortunately is still significant.

Organization of the Paper. Section 2 introduces fully homomorphic encryption and neural networks, the main components of HCNNs. Following that, Section 3 discusses the challenges of adapting convolutional neural networks to the homomorphic domain. Next, we describe the components that were used in implementing HCNNs in Section 4. In Section 5, we report the results of experiments done using our implementation of HCNNs on MNIST and CIFAR-10 datasets. Lastly, we conclude with Section 6 and discuss some of the obstacles that will be faced when scaling HCNNs to larger datasets and deeper networks.

2 Preliminaries

In this section, we review a set of notions that are required to understand the paper. We start by introducing FHE, thereby describing the BFV scheme, an instance of levelled FHE schemes. Next, we introduce neural networks and how to tweak them to become compatible with FHE computation model.

2.1 Fully Homomorphic Encryption

First proposed by Rivest *et al.* [35], (FHE) was envisioned to enable arbitrary computation on encrypted data. FHE would support operations on ciphertexts that translate to functions on the encrypted messages within. It remained unrealized for more than 30 years until Gentry [18] proposed the first construction. The blueprint of this construction remains the only method to design FHE schemes. The (modernized) blueprint is a simple two-step process. First, a somewhat homomorphic encryption scheme that can evaluate its decryption function is designed. Then, we perform bootstrapping, which decrypts a ciphertext using an encrypted copy of the secret key. Note that the decryption function here is evaluated homomorphically, i.e., on encrypted data and the result of decryption is also encrypted.

As bootstrapping imposes high computation costs, we adopt a levelled FHE scheme instead, which can evaluate functions up to a pre-determined multiplicative depth without bootstrapping. We chose the Brakerski-Fan-Vercauteren (BFV) scheme [11, 17], whose security is based on the

Ring Learning With Errors (RLWE) problem proposed by Lyubashevsky *et al.* [34]. This problem is conjectured to be hard even with quantum computers, backed by reductions (in [34] among others) to worst-case problems in ideal lattices.

The BFV scheme has five algorithms (KeyGen, Encrypt, Decrypt, HAdd, HMult). KeyGen generates the keys used in a FHE scheme given the parameters chosen. Encrypt and Decrypt are the encyption and decryption algorithms respectively. The differentiation between FHE and standard public-key encryption schemes is the operations on ciphertexts; which we call HAdd and HMult. HAdd outputs a ciphertext that decrypts to the sum of the two input encrypted messages while HMult outputs one that decrypts to the product of the two encrypted inputs.

We informally describe the basic scheme below and refer to [17] for the complete details. Let $k, q, t > 1$ with $N = 2^k$, t prime and $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1\rangle$, we denote the ciphertext space as $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ and message space as $\mathcal{R}_t = \mathcal{R}/t\mathcal{R}$. We call ring elements "small" when their coefficients have small absolute value.

- $\mathsf{KeyGen}(\lambda, L)$: Given security parameter λ and level L as inputs, choose k, q so that security level λ is achieved. Choose a random element $a \in \mathcal{R}_q$, "small" noise $e \in \mathcal{R}_q$ and secret key $s \in \mathcal{R}_2$, the public key is defined to be $pk = (b = e - as, a)$.
- $\mathsf{Encrypt}(pk, m)$: Given public key pk and message $m \in \mathcal{R}_t$ as input, the encryption of m is defined as $\mathbf{c} = (br' + e' + \lfloor q/t \rfloor m, ar')$, for some random noise $e', r' \in \mathcal{R}_q$.
- $\mathsf{Decrypt}(sk, \mathbf{c})$: Given secret key sk and ciphertext $\mathbf{c} = (c_0, c_1) \in \mathcal{R}_q^2$ as inputs, the decryption of $\mathbf{c}$ is
$$m = \lceil (t/q)(c_0 + c_1 s \bmod q) \rfloor \bmod t.$$
- $\mathsf{HAdd}(\mathbf{c}_1, \mathbf{c}_2)$: Given two ciphertexts $\mathbf{c}_1 = (c_{0,1}, c_{1,1}), \mathbf{c}_2 = (c_{0,2}, c_{1,2})$ as inputs, the operation is simply component-wise addition, i.e. the output ciphertext is $\mathbf{c}' = (c_{0,1} + c_{0,2}, c_{1,1} + c_{1,2})$.
- $\mathsf{HMult}(\mathbf{c}_1, \mathbf{c}_2)$: Given two ciphertexts $\mathbf{c}_1 = (c_{0,1}, c_{1,1}), \mathbf{c}_2 = (c_{0,2}, c_{1,2})$ as inputs, proceed as follows:
  1. (Tensor) compute
  $$\mathbf{c}^* = (c_{0,1}c_{0,2}, c_{0,1}c_{1,2} + c_{1,1}c_{0,2}, c_{1,1}c_{1,2}); \tag{1}$$
  2. (Scale and Relinearize) output
  $$\mathbf{c}' = \lceil \mathsf{Relinearize}(\lceil (t/q)\mathbf{c}^* \rfloor) \rfloor \bmod q. \tag{2}$$

Where $\mathsf{Relinearize}(\mathbf{c}^*)$ is used to shrink the size of $\mathbf{c}^*$ from three back to two terms.

2.1.1 Correctness of the Scheme.

For the scheme to be correct, we require that $\mathsf{Decrypt}(sk, \mathbf{c})$ for $\mathbf{c}$ output from $\mathsf{Encrypt}(pk, m)$, where $(pk, sk = s)$ is a correctly generated key-pair from KeyGen. We characterize when decryption will succeed in the following theorem. Let $\mathbf{c} = (c_0 = \lfloor q/t \rfloor m' + e - c_1 s, c_1)$ be a ciphertext. Then, Decrypt outputs the correct message $m = m'$ if $\|e\|_\infty < q/2t$, where $\|e\|_\infty$ is the largest coefficient of the polynomial $e \in \mathcal{R}_q$.

Proof. Recall that the decryption procedure computes $m = \lceil (t/q)(c_0 + c_1 s) \bmod q \rfloor \bmod t$. Therefore, to have $m = m'$, we first require $c_0 + c_1 s < q$ which means that $\lfloor q/t \rfloor m' + e < q$. Finally, we need the rounding operation to output m' after scaling by t/q which requires that $\|e\|_\infty < q/2t$ since $(t/q) \cdot e$ must be less than $1/2$. □

To see why HAdd works, part of the decryption requires computing

$$\begin{aligned} c_{0,1} + c_{0,2} + (c_{1,1} + c_{1,2})s &= (c_{0,1} + c_{1,1}s) + (c_{0,2} + c_{1,2}s) \\ &= \lceil q/t \rfloor m_1 + e_1 + \lceil q/t \rfloor m_2 + e_2 \\ &= \lceil q/t \rfloor (m_1 + m_2) + e_1 + e_2. \end{aligned}$$

This equation remains correct modulo q as long as the errors are small, i.e. $\|e_1 + e_2\|_\infty < q/2t$. Therefore, scaling by (t/q) and rounding will be correct which means that we obtain the desired message.

For HMult, the procedure is more complicated but observe that

$$\begin{aligned} &(c_{0,1} + c_{1,1}s)(c_{0,2} + c_{1,2}s) = \\ &\qquad c_{0,1}c_{0,2} + (c_{0,1}c_{1,2} + c_{1,1}c_{0,2})s + c_{1,1}c_{1,2}s^2 \end{aligned} \tag{3}$$

This means that we need s as well as s^2 to recover the desired message from $\mathbf{c}^*$. However, with a process called relinearization (Relinearize), proposed by Brakerski and Vaikuntanathan [12] and applicable to the BFV scheme, $\mathbf{c}^*$ can be transformed to be decryptable under the original secret key s.

2.1.2 Computation Model with Fully Homomorphic Encryption.

The set of functions that can be evaluated with FHE are arithmetic circuits over the plaintext ring $\mathcal{R}_t$. However, this is not an easy plaintext space to work with; elements in $\mathcal{R}_t$ are polynomials of degree up to several thousand. Addressing this issue, Smart and Vercauteren [38] proposed a technique to support single instruction multiple data (SIMD) by decomposing $\mathcal{R}_t$ into a product of smaller spaces with the Chinese Remainder Theorem over polynomial rings. For prime $t \equiv 1 \bmod 2N$, $X^N + 1 \equiv \prod_{i=1}^{N}(X - \alpha_i) \bmod t$ for some $\alpha_i \in \{1, 2, \ldots, t-1\}$. This means that $\mathcal{R}_t = \prod_{i=1}^{N} \mathbb{Z}_t[X]/\langle X - \alpha_i \rangle \cong \prod_{i=1}^{N} \mathbb{Z}_t$. Therefore, the computation model generally used with homomorphic encryption is arithmetic circuits with modulo t gates.

For efficiency, the circuits evaluated using the HAdd and HMult algorithms should be levelled. This means that the gates of the circuits can be organized into layers, with inputs in the first layer and output at the last, and the outputs of one layer are inputs to gates in the next layer. In particular, the most important property of arithmetic circuits for FHE is its depth. The depth of a circuit is the maximum number of multiplication gates along any path of the circuit from the input to output layers.

A levelled FHE scheme with input level L can evaluate circuits of at most depth L which affects the choice of parameter q due to noise in ciphertexts. In particular, the HMult operation on ciphertext is the main limiting factor to homomorphic evaluations. By multiplying two ciphertexts results, we have:

$$\begin{aligned} c_{0,1}c_{0,2} &= (\lceil q/t \rfloor m_1 + e_1)(\lceil q/t \rfloor m_2 + e_2) \\ &= (\lceil q/t \rfloor m_1)(\lceil q/t \rfloor m_2) + \\ &\underbrace{\lceil q/t \rfloor m_1 e_2 + \lceil q/t \rfloor m_2 e_1 + e_1 e_2}_{e'}. \end{aligned}$$

Even after scaling by t/q, the overall noise ($\approx t/q \cdot e'$) in the output $\mathbf{c}'$ is larger than that of the inputs, $\mathbf{c}_1$ and $\mathbf{c}_2$. Successive calls to HMult have outputs that steadily grow. Since decryption only succeeds if the error in the ciphertext is less than $q/2t$, the maximum depth of a circuit supported is determined by the ciphertext modulus q. To date, the only known method to sidestep this is with the bootstrapping technique proposed by Gentry [18].

2.2 Neural Networks

A neural network can be seen as an arithmetic circuit comprising a certain number of layers. Each layer consists of a set of nodes, with the first being the input of the network. Nodes in the layers beyond the first take the outputs from a subset of nodes in the previous layer and output the evaluation of an activation function over them. The values of the nodes in the last layer are the outputs of the neural network.

The most widely-used layers can be grouped into three categories:

1. Activation layers: Each node in this layer takes the output, z, of a single node of the previous layer and outputs $f(z)$ for some function z.
2. Convolution-Type layers: Each node in this layer takes the outputs, $\mathbf{z}$, of some subset of nodes from the previous layer and outputs a weighted-sum $\langle \mathbf{w}, \mathbf{z} \rangle + b$ for some weight vector $\mathbf{w}$ and bias b.
3. Pooling layers: Each node in this layer takes the outputs, $\mathbf{z}$, of some subset of nodes from the previous layer and outputs $f(\mathbf{z})$ for some function f.

The functions used in the activation layers are quite varied, including sigmoid ($f(z) = \frac{1}{1+e^{-z}}$), softplus ($f(z) = \log(1 + e^z)$) and ReLU, where

$$\mathsf{ReLU}(z) = \begin{cases} z, & \text{if } z \geq 0; \\ 0, & \text{otherwise.} \end{cases}$$

To adapt neural networks operations over encrypted data, we use the following layers:

- *Convolution (weighted-sum) layer*: at each node, we take a subset of the outputs of the previous layer, also called a filter, and perform a weighted-sum on them to get its output.
- *Average-Pooling layer*: at each node, we take a subset of the outputs of the previous layer and compute the average on them to get its output.
- *Square* layer: each node linked to a single node z of the previous layer; its output is the square of z's output.
- *Fully Connected* layer: similar to the convolution layer, each node outputs a weighted-sum, but over the entire previous layer rather than a subset of it.

3 Homomorphic Convolutional Neural Networks

Homomorphic encryption (HE) enables computation directly on encrypted data. This is ideal to handle the challenges that M/DL face when it comes to questions of data privacy. We call CNNs that operate over encrypted data as Homomorphic CNNs (HCNNs). Although FHE promises a lot, there are several challenges, ranging from the choice of plaintext space to translating neural network operations, that prevent straightforward translation of standard techniques for traditional CNNs to HCNNs. These challenges are described below.

3.1 Plaintext Space

The first problem is the choice of plaintext space for HCNN computation. Weights and inputs of a neural network are usually decimals, which are represented in floating-point. Unfortunately, these cannot be directly encoded and processed in most FHE libraries and thus require special treatment. For simplicity and to allow inference on large datasets, we pack the same pixel of multiple images in a single ciphertext as shown in Figure 1. Note that the BFV scheme can be instantiated such that ciphertexts may contain a number of slots to store multiple plaintext messages. It should be remarked that this packing scheme was first proposed by CryptoNets [16].

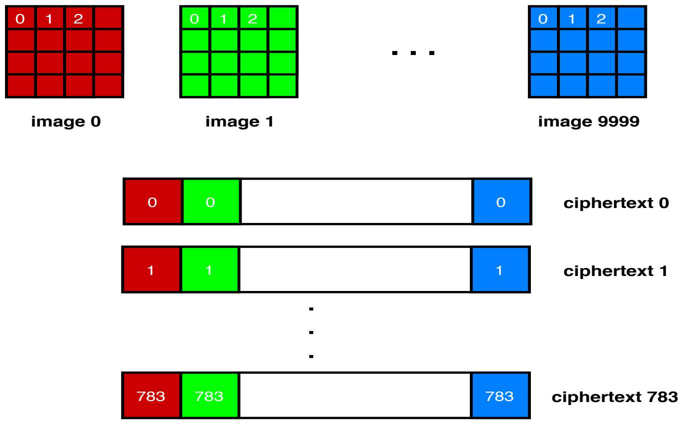


Figure 1: Packing MNIST testing dataset. Ciphertext i contains pixel i from all images.

Encoding into the Plaintext Space. We adopt the *scalar encoding*, which approximates these decimals with integers. It is done by multiplying them with a scaling factor Δ and rounding the result to the nearest integer. Then, numbers encoded with the same scaling factor can be combined with one another using integer addition or multiplication. For simplicity, we normalize the inputs and weights of HCNNs in between $[0, 1]$ and Δ (initially) corresponds to the number of bits of precision of the approximation, as well as the upper bound on the approximation.

Although straightforward to use, there are some downsides to this encoding. The scale factor cannot be adjusted mid-computation and mixing numbers with different scaling factors is not straightforward. For example, suppose we have two messages $\Delta_1 m_1, \Delta_2 m_2$ with two different scaling factors, where $\Delta_1 < \Delta_2$:

$$\Delta_1 m_1 + \Delta_2 m_2 = \Delta_2(m_2 + \Delta_2/\Delta_1 m_1)$$
$$\Delta_1 m_1 \times \Delta_2 m_2 = \Delta_1 \Delta_2(m_1 m_1).$$

Multiplication will just change the scaling factor of the result to $\Delta_1\Delta_2$ but the result of adding two encoded numbers is not their standard sum. This means that as homomorphic operations are done on encoded data, the scaling factor in the outputs increases without a means to control it. Therefore, the plaintext modulus t has to be large enough to accommodate the maximum number that is expected to result from homomorphic computations.

With the smallest scaling factor, $\Delta = 2$, 64 multiplications will suffice to cause the result to potentially overflow the space of 64-bit integers. Unfortunately, we use larger Δ in most cases which means that the expected maximum will be much larger. Thus, we require a way to handle large plaintext moduli of possibly several hundred bits, which is described next.

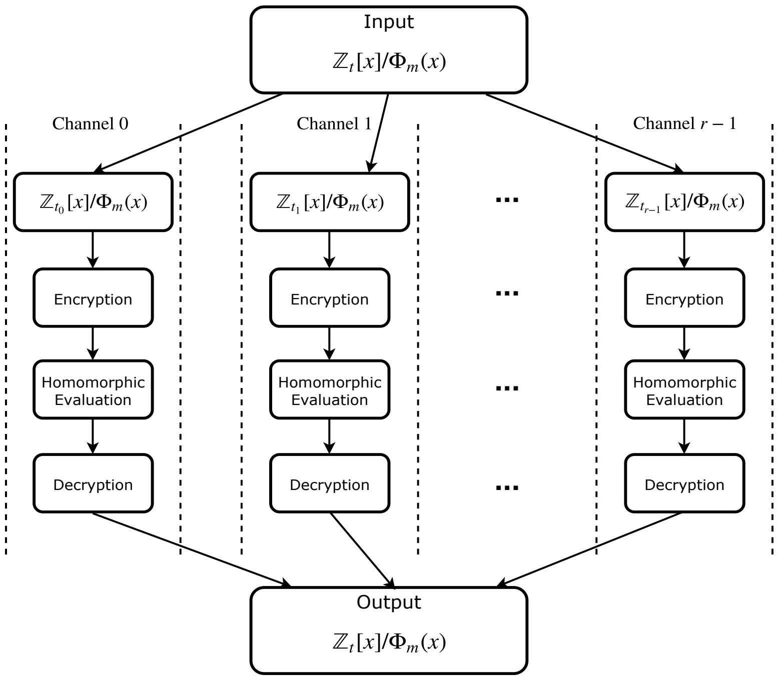

Input
$\mathbb{Z}_t[x]/\Phi_m(x)$
Channel 0
Channel 1
Channel $r-1$
$\mathbb{Z}_{t_0}[x]/\Phi_m(x)$
$\mathbb{Z}_{t_1}[x]/\Phi_m(x)$
$\mathbb{Z}_{t_{r-1}}[x]/\Phi_m(x)$
Encryption
Homomorphic Evaluation
Decryption
Output
$\mathbb{Z}_t[x]/\Phi_m(x)$

Figure 2: Plaintext CRT decomposition for a FHE arithmetic circuit

Plaintext Space CRT Decomposition. One way to achieve this is to use a composite plaintext modulus, $t = \prod_{i=0}^{r-1} t_i$ for some primes $t_0, \ldots, t_{r-1}$ such that t is large enough to accommodate the maximum intermediate result the network may generate. Recall that the Chinese Remainder Theorem (CRT) gives us an isomorphism between $\mathbb{Z}_t$ and $\prod_{i=0}^{r-1} \mathbb{Z}_{t_i}$:

$$\begin{aligned} \mathsf{CRT}: \ \mathbb{Z}_{t_0} \times \cdots \times \mathbb{Z}_{t_{r-1}} &\longrightarrow \mathbb{Z}_t \\ \mathbf{m} = (m_0, \ \ldots, \ m_{r-1}) &\longmapsto m, \end{aligned}$$

where $m_i \in \mathbb{Z}_{t_i}$ and $m \in \mathbb{Z}_t$. The inverse map is:

$$\begin{aligned} \mathsf{ICRT}: \ \mathbb{Z}_t &\longrightarrow \mathbb{Z}_{t_0} \times \cdots \times \mathbb{Z}_{t_{r-1}} \\ m &\longmapsto \mathbf{m} = (m_0, \ \ldots, \ m_{r-1}); \end{aligned}$$

where for any $m \in \mathbb{Z}_t$, we have $\mathsf{CRT}(\mathsf{ICRT}(m)) = m$.

For such moduli, we can decompose any integer $m < t$ into a length-r vector with ICRT. Arithmetic modulo t is replaced by component-wise addition and multiplication modulo the prime t_i for the i-th entry of $\mathbf{m}$. We can recover the output of any computation with CRT.

As illustrated in Figure 2, for homomorphic operations modulo t, we separately encrypt each entry of $\mathbf{m}$ in r FHE instances with the appropriate t_i and perform modulo t_i operations. At the end of the homomorphic computation of function f, we decrypt the r ciphertexts, which gives us $f(\mathbf{m})$. The actual output $f(m)$ is obtained by applying the CRT map to $f(\mathbf{m})$.

3.2 Neural Network Layers

Computation in FHE schemes are generally limited to addition and multiplication operations over ciphertexts. As a result, it is easy to compute polynomial functions with FHE schemes. As with all FHE schemes, encryption injects a bit of noise into the data and each operation on ciphertexts increases the noise within it. As long as the noise does not exceed some threshold, decryption is possible. Otherwise, the decrypted results are essentially meaningless.

Approximating Non-Polynomial Activations. For CNNs, a major stumbling block for translation to the homomorphic domain is the activation functions. These are usually not polynomials, and therefore unsuitable for evaluation with FHE schemes. The effectiveness of the ReLU function in convolutional neural networks means that it is almost indispensable. Therefore, it should be approximated by some polynomial function to try to retain as much accuracy as possible. The choice of approximating polynomial depends on the desired performance of the HCNN. For example, in this work, we applied the square function, $z \mapsto z^2$, which Dowlin *et al.* [16] found to be sufficient for accurate results on the MNIST dataset with a five-layer network.

The choice of approximation polynomial determines the depth of the activation layers as well as its complexity (number of HMults). The depth and complexity of this layer will be $\lceil \log d \rceil$ and $d - 1$ respectively, where d is the degree of the polynomial. However, with the use of scalar encoding, there is another effect to consider. Namely, the scaling factor on the output will be dependent on the depth of the approximation, i.e. if the scaling factor of the inputs to the activation layer is Δ, then the scaling factor of the outputs will be roughly $\Delta^{1+\lceil \log d \rceil}$, assuming that the approximation is a monic polynomial.

Handling Pooling Layers. Similar to activations, the usual functions used in pooling layers, maximum ($\mathsf{max}(\mathbf{z}) = \max_{1 \le i \le n} z_i$), ℓ_2-norm and mean ($\mathsf{avg}(\mathbf{z}) = \frac{1}{n}\sum_{i=1}^{n} z_i$) for inputs $\mathbf{z} = (z_1, \ldots, z_n)$, are generally non-polynomial. Among these, avg is the most FHE-friendly as it requires a number of additions and scaling by a known constant. We note that several works [40, 26] have shown that pooling is not strictly necessary and good results can be obtained without it. We found that pooling layers are not necessary for our MNIST network. On the other hand, pooling gave better accuracy results with the CIFAR-10 network.

Convolution-Type Layers. Lastly, we have the convolutional-type layers. Since these are weighted sums, they are straightforward to compute over encrypted data; the weights can be multiplied to encrypted inputs with $\mathsf{HMultPlain}$ and the results summed with HAdd. Nevertheless, we still have to take care of the scaling factor of outputs from this layer. At first thought, we may take the output scaling factor as $\Delta_w \Delta_i$, multiply the scaling factor of the weights and the inputs, denoted with Δ_w and Δ_i respectively. But, there is actually the potential for numbers to increase in bit-size from the additions done in weighted sums. Recall that when adding two Δ-bit numbers, the upper bound on the sum is $\Delta + 1$ bits long. Therefore, the maximum number that can appear in the worst-case in the convolutions is about $\Delta_w \Delta_i \times 2^{\lceil \log n \rceil}$ bits long, where n is the number of terms in the summands. In practice, this bound is usually not achieved since the summands are almost never all positive. With negative numbers in the mix, the actual contribution from the summation can be moderated by some constant $0 < c < 1$.

4 Implementation

Implementation is comprised of two parts: 1) training on unencrypted data and 2) classifying encrypted data. Training is performed using the 5-layer (for MNIST) and 11-layer (for CIFAR-

10) networks whose details are shown in Table 1 and 2. We use the Tensorpack framework [41] to train the network and compute the model. This part is quite straightforward and can be simply verified by classifying the unencrypted test dataset. For neural networks design, one of the major constraints posed by homomorphic encryption is the limitation of numerical precision of layer-wise weight variables. Training networks with lower precision weights would significantly prevent the precision explosion in ciphertext as network depth increases and thus speed up inference rate in encrypted domain. To this end, we propose to train low-precision networks from scratch, without incurring much loss in accuracy compared to networks trained in floating point precision. Following [42], for each convolutional layer, we quantize floating point weight variables w to k bits numbers w_q using simple uniform scalar quantizer shown below:

$$w_q = \frac{1}{2^k - 1} round(w * (2^k - 1))$$

This equation is a non-differentiable function, we use Straight Through Estimator (STE) [8] to enable the back-propagation. We trained the 5-layer network on MNIST training set with a precision of weights at 2, 4, 8 and 32 bits, and evaluated on MNIST test set with reported accuracy 96%, 99%, 99% and 99% respectively. In view of this, we choose the 4-bit network for the following experiments. It's worth noting that CryptoNets [16] requires 5 to 10 bits of precision on weights to hit 99% accuracy on MNIST test set, while our approach further reduces it to 4 bits and still maintains the same accuracy with both floating-point and scalar encoding. For CIFAR-10, as the problem is more challenging and the network is deeper, our network shown in Table 2 uses 8 bits and achieves 77.80% and 77.55% classification accuracy with the floating-point and scalar encoding, respectively.

The second part is more involved as it requires running the network (with the pre-learned model) on encrypted data. First, we need to fix FHE parameters to accommodate for both the network multiplicative depth and precision. We optimized the scaling factors in all aspects of the HCNN. For the MNIST network, inputs were normalized to $[0, 1]$, scaled by 4 and then rounded to nearest integer. With the low-precision network trained from scratch, we convert the weights of the convolution-type layers to short 4-bit integers, using a small scaling factor of 15; no bias was used in the convolutions. Similarly, inputs to the CIFAR-10 network were normalized to $[0, 1]$ but we used much larger scale factors for the convolution layers. Moreover, padding has been used as it provided better accuracy. The scaling factors for both networks are shown in Tables 1 and 2.

Next, we implement the networks (with scalar encoding) using NTL [37] (a multi-precision number theory C++ library). NTL is used to facilitate the treatment of the scaled inputs and accommodate for precision expansion of the intermediate values during the network evaluation. We found that the largest precision needed is less than (2^{43}) for MNIST and (2^{218}) for CIFAR-10. Note that for MNIST, it is low enough to fit in a single word on 64-bit platforms without overflow. On the other hand, we use the plaintext CRT decomposition to handle the large plaintext modulus required for CIFAR-10. By estimating the maximum precision required by the networks, we can estimate the FHE parameters required by HCNN.

The next step is to implement the network using a FHE library. We implement MNIST HCNN using two FHE libraries: SEAL [1] and GPU-accelerated BFV (A*FV) that is described in [3]. On the other hand, we implement CIFAR-10 HCNN only using A*FV as it is more computationally intensive and would take a very long time. The purpose of implementing MNIST HCNN in SEAL is to facilitate a more unified comparison under the same system parameters and show

Table 1: HCNN architecture for training and testing MNIST dataset with the scale factor used in scalar encoding. Inputs are scaled by 4.

Layer Type	Description	Layer Size	Scale
Convolution	5 filters of size 5×5 and stride $(2, 2)$ without padding.	$12 \times 12 \times 5$	15
Square	Outputs of the previous layer are squared.	$12 \times 12 \times 5$	1
Convolution	50 filters of size 5×5 and stride $(2, 2)$ without padding.	$4 \times 4 \times 50$	15
Square	Outputs of the previous layer are squared.	$4 \times 4 \times 50$	1
Fully Connected	Weighted sum of the entire previous layer with 10 filters, each output corresponding to 1 of the possible 10 digits.	$1 \times 1 \times 10$	15

the superiority of the GPU implementation. In addition, we would like to highlight a limitation in the Residue Number Systems (RNS) variant that is currently implemented in SEAL.

4.1 HCNN Complexity

In this section, we break down the complexity of both HCNNs layers and calculate the total number of operations required for homomorphic evaluation. We also compare our MNIST HCNN with CryptoNets's network [16].

Table 3 shows the computational complexity of each layer in MNIST HCNN and CryptoNets. The convolution and fully connected layers require homomorphic multiplication of ciphertext by plaintext (HMultPlain). Suppose the input to the convolution or fully connected layers is vector $\mathbf{i}$ of length l and the output is vector $\mathbf{o}$ of length m. Let f_w and f_h denote the filter width and height, respectively. The total number of HMultPlain in the convolution layer can be found by $m \cdot f_w \cdot f_h$. The fully connected layer requires $l \cdot m$ HMultPlain operations. On the other hand, the square layer requires $l = m$ ciphertext by ciphertext multiplications (HMult). It should be noted that most of FHE libraries provide an additional procedure for homomorphic squaring (HSquare) which has slightly lower computational complexity compared to HMult, (see Table 8). It can be clearly noticed that HCNN requires much lower number of HMultPlain compared to CryptoNets (46,000 vs 106,625). In CryptoNets, the third layer combines 4 linear layers (2 scaled mean pool, convolution and fully connected layers) for efficiency reasons, whereas it is simply a convolution layer in HCNN. On the other hand, CryptoNets requires less HSquare (945 vs 1,520).

Table 4 shows the computational complexity for CIFAR-10 HCNN. It can be clearly seen that CIFAR-10 HCNN is more computationally intensive compared to MNIST. For instance, 6,952,332 HMultPlain and 57,344 HMult operations are required compared to 46,000 and 1,520, respectively for MNIST HCNN.

4.2 Choice of Parameters

Similar to other cryptographic schemes, one needs to select FHE parameters to ensure that known attacks are computationally infeasible. We denote to the desired security parameter by λ measured in bits. This means that an adversary needs to perform 2^λ elementary (or bit) operations to break the scheme with probability one. A widely accepted estimate for λ in the literature is ≥ 80 bits [39], which is used here to generate the BFV parameters.

Table 2: HCNN architecture for training and testing CIFAR-10 dataset with the scale factor used in scalar encoding. Inputs are scaled by 255.

Layer Type	Description	Layer Size	Scale
Convolution	32 filters of size 3 ÃŮ 3 x 3 and stride (1, 1) with padding.	32 x 32 x 32	10000
Square	Outputs of the previous layer are squared.	32 x 32 x 32	1
Pooling	Average pooling with extent 2 and stride 2.	16 x 16 x 32	4
Convolution	64 filters of size 3 ÃŮ 3 x 32 and stride (1, 1) with padding.	16 x 16 x 64	4095
Square	Outputs of the previous layer are squared.	16 x 16 x 64	1
Pooling	Average pooling with extent 2 and stride 2.	8 x 8 x 64	4
Convolution	128 filters of size 3 ÃŮ 3 x 64 and stride (1, 1) with padding.	8 x 8 x 128	10000
Square	Outputs of the previous layer are squared.	8 x 8 x 128	1
Pooling	Average pooling with extent 2 and stride 2.	4 x 4 x 128	4
Fully Connected	Weighted sum of the entire previous layer with 256 filters	1 x 1 x 256	1023
Fully Connected	Weighted sum of the entire previous layer with 10 filters.	1 x 1 x 10	63

Table 3: MNIST HCNN vs CryptoNets [16] complexity for homomorphic inference

Layer #	MNIST HCNN				CryptoNets			
	Input Neurons	Output Neurons	# of Multiplications		Input Neurons	Output Neurons	# of Multiplications	
			HMultPlain	HMult			HMultPlain	HMult
1	28×28 = 784	5×12×12 = 720	25×720 = 18,000	-	29×29 = 841	5×13×13 = 845	25×845 = 21,125	-
2	5×12×12 = 720	5×12×12 = 720	-	720	5×13×13 = 845	5×13×13 = 845	-	845
3	5×12×12 = 720	4×4×50 = 800	25×800 = 20,000	-	5×13×13 = 845	1×1×100 = 100	100×845 = 84,500	-
4	4×4×50 = 800	4×4×50 = 800	-	800	1×1×100 = 100	1×1×100 = 100	-	100
5	4×4×50 = 800	1×1×10 = 10	10×800 = 8,000	-	1×1×100 = 100	1×1×10 = 10	10×100 = 1,000	-
	Total		**46,000**	1,520	Total		106,625	**945**

Table 4: CIFAR-10 HCNN complexity for homomorphic inference

Layer #	CIFAR-10 HCNN			
	Input Neurons	Output Neurons	No. of Multiplications	
			HMultPlain	HMult
1	32 x 32 x 3 = 3072	32 x 32 x 32 = 32768	589,824	-
2	32 x 32 x 32 = 32768	32 x 32 x 32 = 32768	-	32,768
3	32 x 32 x 32 = 32768	16 x 16 x 32 = 8192	0	-
4	16 x 16 x 32 = 8192	16 x 16 x 64 = 16384	2,594,048	-
5	16 x 16 x 64 = 16384	16 x 16 x 64 = 16384	-	16,384
6	16 x 16 x 64 = 16384	8 x 8 x 64 = 4096	0	-
7	8 x 8 x 64 = 4096	8 x 8 x 128 = 8192	3,308,544	-
8	8 x 8 x 128 = 8192	8 x 8 x 128 = 8192	-	8,192
9	8 x 8 x 128 = 8192	4 x 4 x 128 = 2048	0	-
10	4 x 4 x 128 = 2048	1 x 1 x 256 = 256	457,398	-
11	1 x 1 x 256 = 256	1 x 1 x 10 = 10	2518	-
	Total		6,952,332	57,344

Table 5: HE parameters for MNIST and CIFAR-10 HCNNs with different parameter sets. Depth refers to the supported multiplicative depth and λ denotes the security level in bits.

HCNN	Param ID	N	$\log q$	Plaintext moduli	Depth	λ
MNIST	1	2^{13}	330	5522259017729	4	82
	2	2^{13}	360	5522259017729	5	76
	3	2^{14}	330	5522259017729	4	175
	4	2^{14}	360	5522259017729	5	159
CIFAR-10	5	2^{13}	300	2424833, 2654209, 2752513, 3604481, 3735553, 4423681, 4620289, 4816897, 4882433, 5308417	7	91

In this work, we used a levelled BFV scheme that can be configured to support a known multiplicative depth L, which can be controlled by three parameters: q, t and noise growth. The first two are problem dependent whereas noise growth depends on the scheme. As mentioned in the previous section, we found that t should be at least 43 (resp. 218) bit integer for MNIST (resp. CIFAR-10) to accommodate the precision expansion in HCNN evaluation.

For our HCNNs, 5 (resp. 11) multiplicative depth is required: 2 (resp. 3) ciphertext by ciphertext (in the square layers) and 3 (resp. 8) ciphertext by plaintext (in convolution, pooling and fully connected layers) operations for MNIST and CIFAR-10 HCNNs, respectively. It is known that the latter has a lower effect on noise growth. This means that L needs not to be set to 5 for MNIST and 11 for CIFAR-10. We found that $L = 4$ and 7 are sufficient to run MNIST and CIFAR-10 HCNNs, respectively in A*FV. However, SEAL required higher depth ($L = 5$) to run our MNIST HCNN. The reason is that SEAL implements the Bajard-Enyard-Hasan-Zucca (BEHZ) [7] RNS variant of the BFV scheme that slightly increases the noise growth due to approximated RNS operations. Whereas in A*FV, the Halevi-Polyakov-Shoup (HPS) [20] RNS variant is implemented which has a lower effect on the noise growth. For a detailed comparison of these two RNS variants, we refer the reader to [2].
Having L and t fixed, we can estimate q using the noise growth bounds enclosed with the BFV scheme. Next, we try to estimate N to ensure a certain security level. To calculate the security level, we used the LWE hardness estimator in [5] (commit `76d05ee`).
The above discussion suggests that the design space of HCNN is not limited depending on the choice of the plaintext coefficient modulus t. We identify a set of possible designs that fit different requirements. The designs vary in the number of factors in t (i.e., number of CRT channels) and the provided security level. We provide four sets of parameters for MNIST and one for CIFAR-10. Parameter sets 2 and 4 are shown here to enable running our MNIST HCNN with SEAL. As will be shown later in the subsequent section, SEAL requires a higher q due to the higher noise growth in its underlying FHE scheme. Table 5 shows the system parameters used for each HCNN with the associated security level. Note that we use 10 primes for the plaintext moduli, each of size 22/23 bits, to run the CIFAR-10 HCNN. The product of these small primes gives us a 219-bit plaintext modulus which is sufficient to accommodate any intermediate result in CIFAR-10 HCNN. Note that we have to run 10 instances of CIFAR-10 HCNN to obtain the desired result. However, these instances are embarrassingly parallel and can be run simultaneously.

It is worth noting also that the choice of the primes in the plaintext modulus is not arbitrary especially if one wants to do the inference for multiple images at once. To enable the packed encoding described in Section 3.1, one has to ensure that $2N|(t-1)$.

4.3 HCNN Inference Library

As most deep learning frameworks do not use functions that fit the restrictions of FHE schemes, we designed an inference library using standard C++ libraries that implement some of the CNN layers using only additions and multiplications. Support for arbitrary scaling factors per layer is included for flexibility and allows us to easily define neural network layers for HCNN inference. We give a brief summary of the scaling factor growth of the layers we used in Table 6.

Table 6: Scaling Factor Growth by Layer

Layer Type	Output Scaling Factor
Convolution-Type ($\sum_{i=1}^{n} w_i z_i$)	$\Delta_o = \Delta_w \Delta_i \cdot 2^{c\lceil \log n \rceil}$, for some $0 < c < 1$.
Square Activation ($f(z) = z^2$)	$\Delta_o = \Delta_i^2$.

where Δ_i and Δ_w are the input and weight scaling factors respectively.

In Section 2.2, we introduced several types of layers that are commonly used in designing neural networks, namely activation, convolution-type and pooling. Now, we briefly describe how our library realizes these layers. For convolution-type layers, they are typically expressed with matrix operations but only require scalar additions and multiplications. Our inference library implements them using the basic form, $b + \sum_{i=1}^{n} w_i \cdot z_i$, for input $\mathbf{z} = (z_1, \ldots, z_n)$ and weights $\mathbf{w} = (w_1, \ldots, w_n)$.

For the activation, some modifications had to be done for compatibility with FHE schemes. In activation layers, the most commonly used functions are ReLU, sigmoid ($f(z) = \frac{1}{1+e^{-z}}$) and softplus ($f(z) = \log(1 + e^z)$). These are non-polynomial functions and thus cannot be directly evaluated over FHE encrypted data. Our library uses integral polynomials to approximate these functions; particularly for our HCNN, we used the square function, $f(z) = z^2$, as a low-complexity approximation of ReLU.

The pooling layers used are average-pooling and they are quite similar to the convolution layers except that the weights are all ones and a scale factor that is equal to the reciprocal of the number of averaged values.

4.4 GPU-Accelerated Homomorphic Encryption

The FHE engine includes an implementation of an RNS variant of the BFV scheme [20] that is described in [3, 2]. The BFV scheme is considered among the most promising FHE schemes due to its simple structure and low overhead primitives compared to other schemes. Moreover, it is a scale-invariant scheme where the ciphertext coefficient modulus is fixed throughout the entire computation. This contrasts to other scale-variant schemes that keep a chain of moduli and switch between them during computation. We use a GPU-based BFV implementation as an underlying FHE engine to perform the core FHE primitives: key generation, encryption, decryption and homomorphic operations such as addition and multiplication.

Our FHE engine (shown in Figure 3) is comprised of three main components:

1. Polynomial Arithmetic Unit (PAU): performs basic polynomial arithmetic such as addition and multiplication.

2. Residue Number System Unit (RNSU): provides additional RNS tools for efficient polynomial scaling required by BFV homomorphic multiplication and decryption.

3. Random Number Generator Unit (RNG): used to generate random polynomials required by BFV key generation and encryption.

In addition, the FHE engine includes a set of Look-Up Tables (LUTs) that are used for fast modular arithmetic and number theoretic transforms required by both PAU and RNSU. For further details on the GPU implementation of BFV, we refer the reader to the aforementioned works.

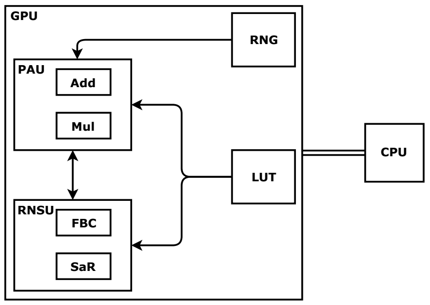


Figure 3: Top-Level Structure of the GPU-Accelerated A*FV Crypto-Processor.

4.4.1 Fitting HCNN in GPU Memory

Another major challenge facing FHE applications is memory requirements due to data expansion after encryption. The ciphertext size can be estimated as $2 * N * \log_2 q$ bits, this is approximately 1.28 MB and 0.59 MB for MNIST and CIFAR-10 HCNNs parameters, respectively. Moreover, due to homomorphic multiplication, ciphertext size may expand $3\times$ its original size due to intermediate computation [2]. As we basically encrypt one pixel - from multiple images - in one ciphertext, this means that in our HCNNs the number of ciphertexts for each layer is equal to the number of pixels in the feature map. For MNIST HCNN, the GPU memory (16 GB) was sufficient to store or generate all the ciphertexts in each layer. On the other hand, the memory was not enough to hold the first convolution layer in CIFAR-10 HCNN. Therefore, we had to process the feature map in batches by uploading parts of the feature map to the GPU memory when needed. An illustrative trajectory of how we span the feature map is shown in Figure 4. Note that we try to minimize the amount of CPU-GPU communication by caching the ciphertexts to be used in subsequent batches.

5 Experiments and Comparisons

In this section, we describe our experiments to evaluate the performance of our HCNNs using the aforementioned designs. We start by describing the hardware configuration. Next, we list the main FHE primitives used to run HCNN alongside benchmarks to provide more insight into their computational complexity. We also present the results of running MNIST HCNN in both SEAL version 2.3.1 and A*FV together with discussion and remarks on performance and noise growth.

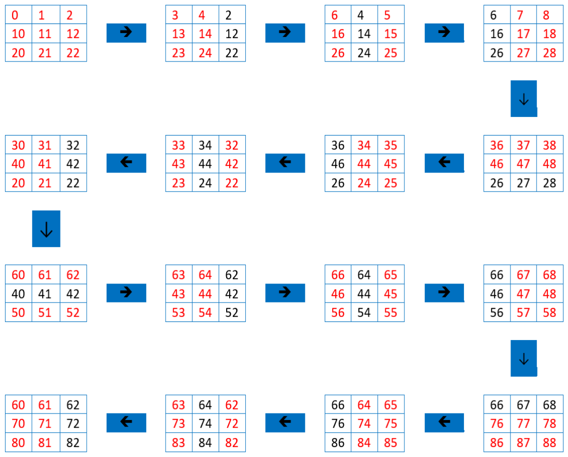


Figure 4: A batched processing of convolution and pooling layers for CIFAR-10 HCNN. Items in red are newly fetched items.

5.1 Hardware Configuration

The experiments were performed on a server with an Intel® Xeon® Platinum 8170 CPU @ 2.10 GHz with 26 cores, 188 GB RAM and an NVIDIA® GPU cluster that includes one V100 and three P100 Tesla cards with 4 × 16 GB on-board memory. Table 7 shows the configuration of the testbed server used for all experiments.

Table 7: Hardware configuration of the testbed servers

Feature	**CPU**	**GPU Cluster**	
Model	Intel(R) Xeon(R) Platinum	V100	P100
Compute Capability	–	7.0	6.0
# of processing units	2	1	3
# Cores (total)	26	5120	3584
Core Frequency	2.10 GHz	1.380 GHz	1.328 GHz
Memory Type	72-bit DDR4	4096-bit HBM2	4096-bit HBM2
Memory Bandwidth	30 GB/sec	732 GB/sec	900 GB/sec
Memory Capacity	187.5 GB	16 GB	3 × 16 GB
PCI-e Bandwidth	16 GB/sec		

5.2 Datasets

MNIST. The MNIST dataset [30] consists of 60,000 images (50,000 in training dataset and 10,000 in testing dataset) of hand-written digits, each is a 28×28 array of values between 0 and 255, corresponding to the gray level of a pixel in the image.

CIFAR-10. The CIFAR-10 dataset [28] consists of 60,000 colour images (50,000 in training dataset and 10,000 in testing dataset) of 10 different classes. Each images consists of $32 \times 32 \times 3$ pixels of values between 0 and 255.

5.3 Micro-Benchmarks

Our HCNNs use 6 FHE primitives: 1) Key generation (KeyGen), 2) Encryption (Enc), 3) Decryption (Dec), 4) Homomorphic addition (HAdd), 5) Homomorphic squaring (HSquare) and 6) Homomorphic multiplication of ciphertext by plaintext (HMultPlain). Table 8 shows these primitives and their latency in milliseconds using SEAL and A*FV on CPU and GPU, respectively. It can be clearly seen that A*FV outperforms SEAL by at least one order of magnitude. On average, A*FV provides 22.36×, 18.67×, 91.88×, 4.40×, 48.07×, 334.56× and 54.59× speedup for KeyGen, Enc, Dec, HAdd, HSquare, HMultPlain and HMult.

The results show that HSquare, which is used in the activation layers, is the most time-consuming operation in our HCNNs. In contrast, both HAdd and HMultPlain, which are used in the convolution and fully connected layers, are very cheap. Note that our HCNNs can be modified to run an encrypted model on encrypted data. This can be done by replacing HMultPlain by HMul. However, this can have a severe effect on performance as HMult is the most expensive primitive in FHE.

Table 8: FHE primitives benchmarks using SEAL and A*FV on CPU and GPU, respectively. Time unit is millisecond. Note that HMult was not used in HCNN.

Function	**Parameter ID**	**SEAL CPU**	**A*FV GPU**	**Speedup**
KeyGen	2	272.142	**12.377**	21.99×
	4	542.920	**21.392**	25.38×
Enc	2	12.858	**0.935**	13.75×
	4	25.991	**1.496**	17.37×
Dec	2	5.171	**0.075**	68.95×
	4	10.408	**0.098**	106.20×
HAdd	2	0.126	**0.052**	2.42×
	4	0.281	**0.054**	5.20×
HSquare	2	69.588	**1.679**	41.45×
	4	138.199	**2.371**	58.29×
HMultPlain	2	7.680	**0.033**	232.73×
	4	15.694	**0.035**	448.40×
HMult*	2	86.270	**2.014**	42.84×
	4	173.167	**2.769**	62.54×

5.4 HCNNs Performance

Table 9 shows the runtime of evaluating our MNIST and CIFAR-10 HCNNs. As mentioned previously, we did not run CIFAR-10 with SEAL as it will take a huge latency and resources. We include the timing of all the aforementioned parameter sets. It can be clearly seen that A*FV outperforms SEAL significantly for all parameter sets. In particular, the speedup factors achieved are 109.86× at 76-bit security level and 182.67× at 159-bit security level. The results show that A*FV is superior at handling large FHE parameters where the maximum speedup is recorded. The amortized time represents the per-image inference time. As the number of slots in ciphertext is equal to N, note that in parameter sets (3 and 4) we can classify the entire testing dataset of MNIST in a single network evaluation. On the other hand, with parameter sets (1 and 2) we can classify 8192 images in a single network evaluation time.

Table 9: Latency (in seconds) of running HCNNs with SEAL and A*FV on multi-core CPU and GPU, respectively. PIT refers to per-image time if packing is used.

HCNN	**Param**	**multi-core CPU**		**GPU**		**Speedup**
	ID	SEAL	PIT	A*FV	PIT	
MNIST_{1G}	1	Failure	–	6.46	0.78×10^{-3}	–
	2	739.90	90.32×10^{-3}	6.73	0.82×10^{-3}	109.86×
	3	Failure	–	8.10	0.81×10^{-3}	–
	4	1563.85	156.38×10^{-3}	8.56	0.85×10^{-3}	182.67×
CIFAR-10_{1G}	5	–	–	553.89	67.61×10^{-3}	–
CIFAR-10_{4G}	5	–	–	304.43	37.162×10^{-3}	–

The table also includes the latency of running our CIFAR-10 HCNN using A*FV. We show the results of running our CIFAR-10 HCNN on 1 and 4 GPU cards. The latency shown here is per 1 plaintext modulus prime, i.e., 1 CRT channel. Note that we use 10 primes to evaluate CIFAR-10 HCNN. As our HCNNs will typically be hosted by the cloud, one may assume that 10 machines can evaluate CIFAR-10 HCNN in 304.430 seconds.

The results also show the importance of our low-precision training which reduced the required precision to represent MNIST HCNN output. This allows running a single instance of the network without plaintext decomposition, i.e., single CRT channel. We remark that CryptoNets used higher precision training and required plaintext modulus of higher precision (2^{80}). Therefore, they had to run the network twice using two CRT channels.

We also note that our timing results shown here for SEAL are much higher than those reported in CryptoNets (570 seconds at 80-bit security). This can be attributed to the following reasons: 1) CryptoNets used the YASHE′ levelled FHE scheme which is known to be less computationally intensive compared to BFV that is currently implemented in SEAL [31]. It should be remarked that YASHE′ is no more considered secure due to the subfield lattice attacks [4], and 2) CryptoNets include a smaller number of HMult as shown in Table 3.

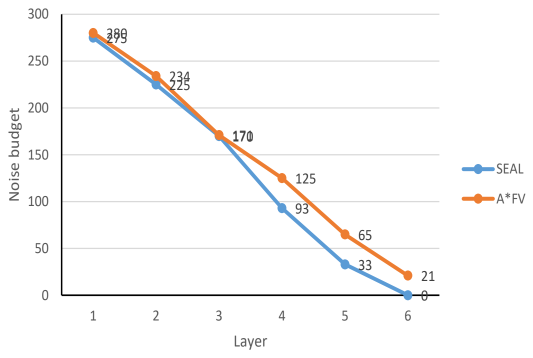
Lastly, we compare our best results with the currently available solutions in the literature. Table 10 shows the reported results of previous works that utilized FHE to evaluate HCNNs on different datasets. As we can see, our solution outperforms all solutions in total and amortized time. For instance, our MNIST HCNN is 70.29×, 3.53× and 4.83× faster than CryptoNets, E2DM and Faster CryptoNets, respectively. Note that E2DM classifies 64 images in a single evaluation. Similarly, our CIFAR-10 HCNN is 3.83× and 7.35× faster than CryptoDL and Faster CryptoNets, respectively.

5.5 Noise Growth

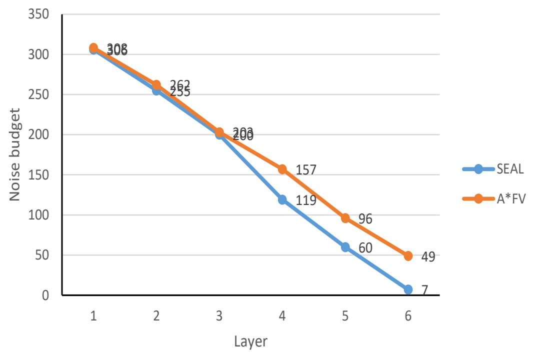
In this experiment, we show the noise growth behaviour in both SEAL and A*FV. We recall that SEAL version (2.3.1) implements the BEHZ [7] RNS variant of the BFV scheme. On the other

Table 10: Comparison of running time (seconds) between prior FHE-based HCNNs and our HCNNs.

Model	Runtime		λ	Dataset	Platform
	Total	Amortized time			
CryptoNets [16]	570	69.580×10^{-3}	80	MNIST	CPU
E2DM [24]	28.590	450.0×10^{-3}	80	MNIST	CPU
Faster CryptoNets [15]	39.100	39.100	128	MNIST	CPU
A*FV	8.100	0.008×10^{-3}	175	MNIST	GPU
CryptoDL [21]	11686	–	80	CIFAR-10	CPU
Faster CryptoNets [15]	22372	22372	128	CIFAR-10	CPU
A*FV	3044	0.372	91	CIFAR-10	GPU



(a) Parameter Set 3



(b) Parameter Set 4

Figure 5: Noise budget (in bits) and how it is consumed by SEAL and A*FV

hand, A*FV implements another RNS variant known as the HPS [20]. Although both variants implement the same scheme, it was found by Al Badawi *et al.* [2] that these variants exhibit different noise growth behaviour. Figure 5 shows the noise growth behaviour in both SEAL and A*FV for the parameter sets 3 and 4. The vertical axis represents the noise budget which can be viewed as the "signal-to-noise" ratio. Once the noise budget reaches 0 in a ciphertext, it becomes too noisy to compute further or to decrypt successfully. It should be remarked that the noise budget can be only calculated given the secret key. As seen in the figure, parameter set 3 is not sufficient to provide SEAL with sufficient noise budget to evaluate the MNIST HCNN. The ciphertexts generated by the fully connected layer include noise budget 0. Although no further computation is required after the fully connected layer, decryption fails due to the high noise level. On the other hand, A*FV has lower noise budget consumption rate that is sufficient to run MNIST HCNN with some left out noise budget (21 bits in parameter set 3 and 49 bits in parameter set 4). Parameter set 4 provides higher noise budget that is sufficient to run the MNIST HCNN in both SEAL and A*FV successfully.

6 Conclusions

In this work, we presented a fully FHE-based CNN that is able to homomorphically classify the encrypted images with FHE. The main motivation of this work was to show that privacy-preserving deep learning with FHE is dramatically accelerated with GPUs and offers a way towards efficient M/DLaaS. Our implementation included a set of techniques such as low-precision training, unified training and testing network, optimized FHE parameters and a very efficient GPU implementation to achieve high performance. We managed to evaluate our MNIST HCNN in one CRT channel in contrast to previous works that required at least two channels. Our solution achieved high security level (> 80 bit) and reasonable accuracy (99%) for MNIST and (77.55%) for CIFAR-10. In terms of performance, our best results show that we could classify the entire testing dataset in 6.46 and 3044 seconds for MNIST and CIFAR-10 respectively, with per-image amortized time (0.78 milliseconds) and (371 milliseconds), respectively.

In its current implementation, our HCNNs have adopted the simple encoding method of packing the same pixel of multiple images into one ciphertext, as described in Section 3.1. This packing scheme is ideal for applications that require the inference of large batches of images which can be processed in parallel in a single HCNN evaluation. Other application may have different requirements such as classifying 1 or a small number of images. For this particular case, other packing methods that pack more pixels of the same image in the ciphertext can be used. As future work, we will investigate other packing methods that can fit a wider range of applications. Moreover, we will target more challenging problems with larger datasets and deeper networks.

In addition, we noticed that training deep CNNs using polynomially-approximated activation functions does not maintain high prediction accuracy. Further research is required to find the best methods to approximate activation functions in FHE.

Acknowledgments

This work was supported by the Institute for Infocomm Research (I2R) - A*STAR and the National University of Singapore.

References

[1] Simple Encrypted Arithmetic Library (release 2.3.1). `http://sealcrypto.org`, 2017. Microsoft Research, Redmond, WA.

[2] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. Cryptology ePrint Archive, Report 2018/589, 2018.

[3] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. 2018(2):70–95, 2018.

[4] Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. pages 153–178, 2016.

[5] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[6] Louis J. M. Aslett, Pedro M. Esperança, and Chri. C. Holmes. Encrypted statistical machine learning: new privacy preserving methods. *ArXiv e-prints*, 2015.

[7] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[8] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, 2013.

[9] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. pages 45–64, 2013.

[10] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. pages 483–512, 2018.

[11] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. pages 868–886, 2012.

[12] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. pages 97–106, 2011.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. pages 3–33, 2016.

[15] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *CoRR*, abs/1811.09953, 2018.

[16] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. Technical report, February 2016.

[17] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[18] Craig Gentry. Fully homomorphic encryption using ideal lattices. pages 169–178, 2009.

[19] Thore Graepel, Kristin Lauter, and Michael Naehrig. ML confidential: Machine learning on encrypted data. pages 1–21, 2013.

[20] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. Cryptology ePrint Archive, Report 2018/117, 2018.

[21] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *CoRR*, abs/1711.05189, 2017.

[22] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N. Wright. Privacy-preserving machine learning as a service. *PoPETs*, 2018(3):123–142, 2018.

[23] Yashpalsinh Jadeja and Kirit Modi. Cloud computing-concepts, architecture and challenges. In *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*, pages 877–880. IEEE, 2012.

[24] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 2018.

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

[26] Konstantinos Kamnitsas, Christian Ledig, Virginia F.J. Newcombe, Joanna P. Simpson, Andrew D. Kane, David K. Menon, Daniel Rueckert, and Ben Glocker. Efficient multi-scale 3d cnn with fully connected crf for accurate brain lesion segmentation. *Medical Image Analysis*, 36:61 – 78, 2017.

[27] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[28] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105. Curran Associates Inc., 2012.

[30] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits, 1998.

[31] Tancrede Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. In *International Conference on Cryptology in Africa*, pages 318–335. Springer, 2014.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[33] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. pages 619–631, 2017.

[34] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. pages 1–23, 2010.

[35] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 4(11):169–180, 1978.

[36] John M Rushby. *Design and verification of secure systems*, volume 15. ACM, 1981.

[37] Victor Shoup et al. Ntl, a library for doing number theory, version 5.4, 2005.

[38] N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014.

[39] Nigel P Smart, Vincent Rijmen, B Gierlichs, KG Paterson, M Stam, B Warinschi, and G Watson. Algorithms, key size and parameters report. *European Union Agency for Network and Information Security*, pages 0–95, 2014.

[40] J.T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR (Workshop Track)*, 2015.

[41] Yuxin Wu et al. Tensorpack. `https://github.com/tensorpack/`, 2016.

[42] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.