REDsec: Running Encrypted DNNs in Seconds

Lars Folkerts, Charles Gouert, Nektarios Georgios Tsoutsos
{folkerts, cgouert, tsoutsos}@udel.edu
University of Delaware

Abstract

Machine learning as a service (MLaaS) has risen to become a prominent technology due to the large development time, amount of data, hardware costs, and level of expertise required to develop a machine learning model. However, privacy concerns prevent the adoption of MLaaS for applications with sensitive data. One solution to preserve privacy is to use fully homomorphic encryption (FHE) to perform the ML computations. FHE has great power to protect sensitive inputs, and recent advancements have lowered computational costs by several orders of magnitude, allowing for practical applications to be developed. This work looks to optimize FHE-based private machine learning inference by leveraging ternary neural networks. Such neural networks, whose weights are constrained to {-1,0,1}, have special properties that we exploit in this work to operate efficiently in the homomorphic domain. We introduce a general framework that takes an input model, performs plaintext training, and efficiently evaluates private inference leveraging FHE. We perform inference experiments with the MNIST, CIFAR-10, and ImageNet datasets and achieve private inference speeds of only 1.7 to 2.7 orders of magnitude slower compared to their plaintext baseline.

I. Introduction

The rapid growth of cloud computing services has amplified concerns about the need for data privacy. Users of these services trust their personal data to the cloud for storage and computation, which can put their privacy at risk. For instance, a curious cloud service provider can read the sensitive user data, since it's stored on their servers. This can potentially allow them to learn proprietary secrets as well as personal data (such as health records) to sell to advertisers [1]. In addition, cyberattacks can be mounted against cloud servers and data stored on these hosts can be leaked or stolen [2], [3]. Attackers are beginning to set their sights on these servers as more and more users take advantage of cloud services and outsource valuable data. Therefore, direct attacks targeting cloud datacenters are becoming increasingly common [4].

Because of these security threats and the fact that a large number of organizations and individuals continue to adopt the cloud computing paradigm, it is essential to provide security guarantees for all

forms of outsourced computation, which each come with their own set of unique challenges. This work focuses on a specific and emergent case of cloud computing known as machine learning as a service (MLaaS) [5]. In this scenario, a cloud service provider has a trained network (often with private weights) on their servers and allows users to upload their own data for classification purposes. For example, in medical research, many new machine learning algorithms are being developed to process medical images and cloud service providers could develop and launch networks using the MLaaS paradigm for users [6]. However, due to legal and regulatory issues surrounding privacy and IP concerns (e.g., HIPAA [7]), the use of cloud computing for these applications remains unexplored. To mitigate this problem, special considerations need to be in place to enable users to securely upload their data to the cloud and receive provable guarantees about their privacy during processing.

The most common way to secure user data and protect confidentiality is through the use of encryption schemes such as AES [8]. While this successfully prevents attackers and the cloud from viewing the data, it also severely limits the usefulness during processing. In particular, no meaningful work can be done with the exception of storage; in other words, common encryption algorithms do not allow executing algorithms on encrypted data, such as those required for MLaaS and other cloud computing scenarios [9]. Luckily, there are a few state-of-the-art cryptographic techniques that allow for purposeful computation to be done on encrypted data while still maintaining confidentiality. The current strategies for privacy preserving MLaaS mainly involve two techniques: homomorphic encryption (HE) [10] and multiparty computation (MPC) [11].

Homomorphic encryption encompasses a special class of ciphers that all share an incredible property: the ability to perform meaningful computations directly on ciphertexts, which, in turn, manipulates the underlying plaintext data in exactly the same way. This capability allows users to outsource secure ciphertexts to a third party cloud service provider to execute algorithms, such as neural network inference, on the encrypted data without any knowledge of the data itself. After the computation is complete, the cloud will send the encrypted result back and users can decrypt to get the output of the algorithm on their original inputs. This type of encryption comes in three flavors: partial HE (PHE), leveled HE (LHE), and fully HE (FHE) that differ in the number and types of operations that can be done on encrypted data.

Many previous works aiming to solve privacy preserving MLaaS utilize LHE, which allows performing arbitrary operations on ciphertext data for *only a predefined, bounded number of times*. Pure LHE-based MLaaS frameworks, such as CryptoNets [12] and CHET [13], allow machine learning inference on encrypted data to be done entirely by the cloud with no interaction from the user except the secure upload of encrypted inputs and downloading the final encrypted result. One of the key principles of HE

is the concept of noise, which accumulates in ciphertexts during every operation and is necessary to guarantee security. If too much noise accumulates in the ciphertexts, the user will be unable to decrypt the data. To accommodate for this, the depth of the computations in LHE must be known beforehand to allocate the proper noise budget and to ensure a correct decryption 100% of the time. In order to increase the noise budget, either the security level must be decreased, or different parameter sets must be selected that result in progressively slower execution times. For complex algorithms that perform many computations with the same data repeatedly, *LHE does not scale* and becomes incredibly inefficient in terms of both speed and memory overheads. For deep neural networks that operate on complex datasets, such as CIFAR-10, this approach is not feasible. In fact, most works using LHE are optimized only for small and simple networks for the MNIST dataset.

To compensate for this, other works have employed FHE which builds upon LHE and adds a "bootstrapping" mechanism that allows the noise in ciphertexts to be reduced when it reaches a certain threshold. This procedure is considered costly and is generally the bottleneck of fully homomorphic operations. However, for complex algorithms, this approach is still more efficient than choosing increasingly larger parameters for LHE. For example, FHE-DiNN [14] employs fully homomorphic encryption to conduct private inference for a tiny neural network, and this approach only demonstrates how to evaluate fully-connected layers. In this case, the authors modify the bootstrapping algorithm adding the capability to automatically compute an activation function on the encrypted data for almost no extra cost. While this is a noteworthy optimization that has come to be known as "programmable bootstrapping" [15], it must be done on all active neurons at least once a layer (in order to evaluate the activation function), even if the ciphertexts have low noise. This results in an overall slower execution time, as the bootstrapping procedure is invoked more times than strictly necessary.

One potential limitation of homomorphic encryption that affects all of the above approaches is the issue of branching on encrypted data. Because the third party carrying out the encrypted computation has no information about what the underlying plaintext of the user is, there is no way for the server to make a runtime decision based on an encrypted value. This is known as the termination problem [16], which is a reference to the fact that servers cannot rely on *early termination of loops* that depend of encrypted conditions and return an answer. For instance, a privacy-preserving search algorithm for a database with homomorphically encrypted records must search the entire database and combine the encrypted results in every query. As a result, for certain encrypted computations where all branch conditions are based on ciphertexts, the servers end up with the worst-case complexity.

The second approach for privacy preserving MLaaS aims to solve the termination problem using

multiparty computation, which involves multiple entities performing functions jointly over their private data. With this approach, no single entity is capable of seeing the data of other parties involved in the computation. A number of popular private MLaaS solutions incorporate both MPC and HE constructions, such as Gazelle [17], Cheetah [18] and MiniONN [19]. These frameworks use LHE or PHE for linear operations on the cloud (e.g., convolutions) and MPC in the form of garbled circuits [20] for non-linear operations (e.g., ReLU activations) that require branch decisions and help from the data owner. In these solutions, the cloud still maintains control over the convolution weights, making them transparent to the user. However, in these solutions, *the user must be kept actively engaged during the computation*, which limits the usefulness and feasibility of these solutions. In addition, there is a large communication overhead between the user and cloud, as data used for MPC computations must be continuously uploaded and downloaded.

Because LHE realistically only supports inference for small neural network, and MPC requires the user to take part in the machine learning computations, in this work we adopt FHE to facilitate inference for *arbitrary* neural networks. While bootstrapping remains the bottleneck of fully homomorphic operations, several works have accelerated the procedure dramatically [21], [22], [23], [24]. In addition, the FHE evaluation can be accelerated even further with GPUs, achieving more than an order of magnitude speedup over a CPU. Even with all of these techniques, the cost of a bootstrap remains much higher than other operations on ciphertexts and minimizing the invocations of the procedure is critical for fast evaluation. Contrary to prior works, our REDsec framework employs ternary neural networks (TNNs) and treats individual ciphertexts in the network as binary bits, which allows us to employ optimizations such as computing *a sign function* for very little cost and no noise penalty. In addition, we adopt a strategy known as *bridging* to convert ciphertexts to the integer domain and accelerate addition operations [25]. These insights allows us to optimize all layer types to require the bare minimum number of bootstraps in order to achieve significantly faster inference speeds.

Our contributions can be summarized as follows:

- An optimal order and computation structure of ternary neural networks to accommodate for an efficient fully homomorphic implementation;
- Major improvements to cuFHE, a state-of-the-art library for GPU accelerated HE operations, including leveled operations, encryption of constants, and robust support for multiple GPUs;
- A detailed analysis of neural network structure to determine the most optimal times to perform costly bootstrapping procedures to refresh the noise;

- An end-to-end system to construct arbitrary neural network architectures, including a UI for users to build their model and a bespoke compiler to generate the training code in TensorFlow and encrypted inference code in C++/CUDA.

Roadmap: In Section II, we provide an overview of homomorphic encryption and machine learning concepts as well as our adopted threat model. Section III provides an overview of REDsec and Section IV provides specific implementation details. Section V details our experimental evaluations and analysis of results. Lastly, Section VI provides comparisons with prior works and Section VII concludes the paper.

II. Preliminaries

A. Learning With Errors

Learning with Errors (LWE) [26], and its variant called Ring-LWE [27], is the hard problem that many homomorphic encryption schemes and other lattice-based encryption algorithms rely on for their security. In turn, LWE derives its hardness assumptions from other important problems in both coding and lattice theory [28], [29]. Solving the LWE problem essentially requires recovering a function from a set of noisy samples. The difficulty of solving this problem, and hence the level of security of schemes based on LWE, is proportional to the magnitude of random noise injected into the original samples. The Ring-LWE problem is an extension of LWE in the domain of polynomials over finite fields.

Adapting this problem to cryptographic applications is relatively straightforward: encryption keys and ciphertexts are injected with noise to hinder cryptanalysis. In the case of homomorphic encryption, there is a tradeoff between the level of injected noise (and hence the security level) and the speed of HE computation. The lower the magnitude of noise, the more homomorphic additions and multiplications can be conducted before measures must be taken (e.g., bootstrapping) to reduce the noise to a secure, but manageable level.

B. Homomorphic Encryption

Cryptographic schemes that support meaningful computation directly on ciphertexts and result in a valid encryption of the result of the computation are deemed homomorphic. However, not all homomorphic encryption schemes are created equal, and they are classified into three categories depending on the type and number of operations that can be computed on ciphertexts.

1) Partially Homomorphic Encryption: The "weakest" category of HE is partially homomorphic encryption (PHE) and was naturally the first realization of homomorphic encryption in general. Cryptosystems such as RSA [30], ElGamal [31], and Paillier [32] fall into this category and allow for only one

of two basic arithmetic operations on ciphertexts: either addition or multiplication. A benefit of this type of homomorphic encryption is that operations are fast and there is no noise accumulation because these schemes do not derive their security from the previously defined LWE problem. The consequence of this is that PHE allows for unlimited additions *or* unlimited multiplications on encrypted data, but not both. In the context of privacy-preserving machine learning, PHE is often combined with MPC to make up for its computational shortcomings for both neural network training and inference [17].

2) Leveled Homomorphic Encryption: Contrary to PHE, both LHE and FHE schemes allow for arbitrary algorithms to be evaluated on encrypted data because both are capable of supporting encrypted additions and multiplications, which form a functionally complete set of operations. However, unlike PHE, LHE is not capable of executing an unbounded number of encrypted operations because of noise accumulation. This fact also makes LHE far more difficult to harness effectively, since complex encryption parameters (e.g., the number of primes in the ciphertext moduli chain, the degrees of various polynomials, and the standard deviation of injected noise) must be carefully tweaked and optimized for both security and the number of operations required by the algorithm being implemented. There is a complicated balancing act between the security level, speed, and noise threshold that must be taken into account for each application in which LHE is used. While this balancing act also exists for FHE, it is generally independent of the actual algorithm being carried out on encrypted data. For LHE, the more operations required for an application, the slower (or less secure) leveled homomorphic operations become in general.

3) Fully Homomorphic Encryption: FHE was realized for the first time in 2009 by Craig Gentry with the introduction of the bootstrapping theorem [10]. As mentioned prior, this technique is used to reduce the noise in ciphertexts to allow for arbitrary encrypted operations. Before bootstrapping, the only way to eliminate the noise in a homomorphic ciphertext was to send it back to the user, have them decrypt, and finally re-upload a fresh ciphertext with minimal noise. The bootstrapping procedure converts this concept to the encrypted domain by having the user provide the cloud with *an encryption of the secret key*. The cloud can then use this key to perform a decryption procedure homomorphically; however, instead of plaintext, the result will be a new encryption of the plaintext with greatly reduced noise. By keeping track of noise growth in ciphertexts and applying the bootstrapping procedure when needed, FHE is capable of computing an infinite number of additions and multiplications on ciphertext data.

C. Contemporary HE Libraries

There are a number of HE libraries to choose from that all offer certain advantages and disadvantages. The first widely available open-source library is called *HElib* and implements the BGV [33] and CKKS

[34] homomorphic cryptosystems. This library treats individual ciphertexts as integers (or approximate numbers) and supports both multiplication and addition operations on ciphertext data. Typically, this library is used for LHE even though it includes FHE support due to particularly slow bootstrapping speeds.

Another popular library called *SEAL* was created by Microsoft and offers *leveled* versions of the BFV [35] and CKKS cryptosystems. Similar to HElib, ciphertexts represent either integers or approximate numbers. SEAL is commonly used to build small privacy-preserving neural network inference applications as it provides an intuitive API and is generally more friendly and easier to configure than other HE libraries.

Both HElib and SEAL are solid options for LHE, but the former is not feasible for FHE and the latter does not include bootstrapping, and thus does not provide FHE support. The *FHEW* [23] scheme, which itself is derived from the GSW cryptosystem [36], takes a completely different approach to HE from these two libraries and greatly improves the speed of bootstrapping. In FHEW, ciphertexts represent individual bits of plaintext values and the operations exposed to users take the form of logic gate operations. As a result, algorithms implemented using FHEW must be in the form of (virtual) digital circuits; for instance, to add two encrypted bytes, one must implement an 8-bit homomorphic adder circuit.

While FHEW boasts bootstrapping speeds of less than a second, *TFHE* expands upon and evolves FHEW's approach to achieve even more efficient bootstrapping capabilities. Similar to FHEW, TFHE treats ciphertexts as encryptions of single bits and provides a logic gate API for users to construct arbitrary algorithms as circuits. TFHE is capable of evaluating a single gate followed by a bootstrapping procedure in 13 milliseconds, with the exception of a homomorphic multiplexer gate that takes approximately double the time of the other gates. Due to its incredibly fast bootstrapping speeds, many privacy-preserving machine learning frameworks that leverage FHE use this library. In turn, REDsec also allows users to select TFHE as the underlying crypto library as it remains the fastest and most feasible option for FHE on CPUs. However, the *cuFHE* [37] and *nuFHE* [38] GPU libraries port the TFHE scheme to CUDA and are capable of accelerating the bootstrapping procedure even further, by over an order of magnitude. To the best of the authors' knowledge, these GPU libraries provide the fastest bootstrapping speeds of any open-source library and boast approximately identical speeds as each other (on the same GPUs and in NTT mode). In its fastest configuration, REDsec employs our heavily modified version of cuFHE to evaluate any homomorphic circuit.

D. Binary Neural Networks

1) Concepts: Binary neural networks (BNNs) constrain weights and/or values to {-1,1}. They are primarily researched as a way to store small weight files on mobile devices, as each {-1,1} weight can be represented as a bit {0,1} [39], [40]. Often, the BNN activation function is a sign function:

$$sign(x) = \begin{cases} +1 & \text{iff } x > 0, \\ -1 & \text{otherwise,} \end{cases} \tag{1}$$

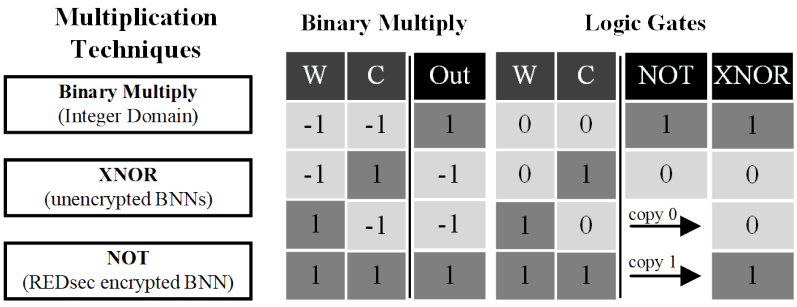
which converts the post-convolution values back to binary. We refer to this class of activation functions as binary activations [39], [40], [41]. Binary neural networks have many advantages that improve the speed of computation, rendering BNNs less costly in terms of memory and execution time compared to full precision networks [39], [40], [41]. Since all of the weights and values are bits, this network is an ideal candidate to run with the TFHE cryptosystem.

There are several works expounding on how to train binary neural networks for quick convergence [41]. In particular, training a binary neural network is an interesting problem, since the gradient for the sign function is undefined. Thus, much of the work on BNNs centers around the problem of picking a suitable gradient function during backwards propagation for training the network [41]. We remark that our paper does not focus on these different implementations, although many are available in the Larq library, which we leverage for our REDsec implementation [42].

TABLE I. **POPULAR NETWORK ARCHITECTURES FOR ALEXNET:** WEIGHT FORMAT, ACTIVATION FUNCTIONS AND REPORTED ACCURACY.

Network	Weights	Activation	Top-1	Top-5
AlexNet [43]	Full Precision	Full Precision	57.1%	80.2%
Binary Weight (BWN) [40]	Binary	Full Precision	56.8%	79.4%
XNOR-net [40]	Binary	Mixed	44.2%	69.2%
BinaryAlexNet [44]	Binary	Binary	36.3%	61.5%
Hybrid Binary (HBN) [45]	Binary	Mixed	48.6%	72.1%
Benn [46]	Binary	Binary	54.3%	N/A

There are also many works that improve network architectures for high accuracy [41]. Indeed, there is a difference between binary weight networks (BWNs) with integer activation functions, and binary-weight/binary-activation networks [42], [41]. This trade off was first explored in the XNOR-net paper [40], and binary weight networks can have similar accuracy to full precision networks when trained properly. Nevertheless, binary-weight/binary-activation networks do receive some accuracy degradation



(a) Strategies (b) Truth Tables

Fig. 1. **Integer to Logical Space:** The core idea behind BNNs is that a complex multiplication operation reduces to an XNOR gate, with the {-1,1} in the integer domain mapping to {0,1} in the XNOR truth table. In the encrypted domain, when the weight (W) is known to the server, REDsec applies either the NOT gate (if W=0) or a copy operation (if W=1) to the ciphertext (C) instead of the noisy and expensive homomorphic XNOR gate.

since information is lost in the binary sign activation function. Likewise, recent works explore hybrid techniques to boost accuracy and still have binary weights and activations [41], [45]; for example, most of the accuracy loss can be mitigated by keeping full-precision pixel values at only a few of the middle layers [47]. Another useful technique is binary network ensembles, where multiple binary neural networks are trained and return a result to the user. In this case, users consolidate these results to improve performance [46]. A comparison of these different strategies is summarized in Table I.

Ternary neural networks is another promising technique for accuracy improvement, which we find to be very effective for our work. Specifically, ternary neural networks offer the possibility of having an additional zero weight: {-1,0,1} [48]. This optimization comes with an increase in accuracy, but since it incurs moderate memory and computation overheads, many discrete neural network implementations overlook this feature [48], [41]. However, this cost is effectively negligible when working with encrypted neural networks, making ternary networks a lucrative alternative to binary neural networks for our system.

2) Multiplication: The concept of binary multiplication was first adapted in BinaryNet [39]. The basic idea is that a multiplication of values in {-1, 1} is equivalent to an XNOR of values in {0, 1}. While this concept makes binary-weight/binary-activation networks easier to implement on FPGAs [49], [41], encrypted XNOR is noisy and expensive. Thus, in REDsec we exploit the fact that weights are know to the server to further reduce the cost of multiplication to a very efficient homomorphic NOT or copy operation (Fig. 1). REDsec also supports integer-value/binary-weight multiplication using a similar logic: multiplication by -1 is the 2's complement, where all bits are flipped and one is added to the result. The bit flip can also be accomplished by an NOT gate, while the plus one can be implemented using a homomorphic add operation or an incrementer circuit:

$$-x = \overline{x} + 1. \tag{2}$$

For ternary neural networks, multiplication by a 0 weight is 0 regardless of the input. Therefore, we do not need to process the input in our calculations, but need to adjust for the zero valued result in the convolution step [48].

3) Convolution: Building on binary multiplication, convolution is a series of multiply-adds. Since XNOR replaces multiplication, this problem reduces to XNOR-bitcount. When the convolution has binary inputs, the bitcount and XNOR operations are not exact representations, since XNOR has outputs $\{0,1\}$ instead of $\{-1,1\}$. The different representations are related based on the number of multiplications M.

$$\textbf{Bitwise Representation: } b = (i + M) \div 2 \tag{3}$$

$$\textbf{Integer Representation: } i = (2 \cdot b) - M \tag{4}$$

For ternary neural networks in the bitwise domain, we need to take into account the 0 values. These zero values actually correspond to 0.5 in the bitwise domain. Thus, when building a network, we can count the number of zero values and divide by two to get an offset.

$$\textbf{Ternary Weight Offset} = \tfrac{1}{2} \sum w \in W \mid (w == 0) \tag{5}$$

Since the sign activation function follows convolution, we can incorporate the count in the sign step, rounding to the nearest number. Likewise, when the convolution has integer inputs, our 2's complement method can be used to represent integers directly. Here, zero valued weights just ignore the input directly.

4) Sign Function: Combining the sign equation (Eq. (1)) with the integer-bitwise conversion (Eq. (4)), we get the following result [40], [41]:

$$sign(i) = sign(2 \cdot b - M) = \begin{cases} +1 & \text{if } (2 \cdot b - M) > 0, \\ -1 & \text{otherwise;} \end{cases} \tag{6}$$

$$bitsign(b) = \begin{cases} 1 \text{ iff } b > \frac{M}{2}, \\ 0 \text{ otherwise.} \end{cases} \tag{7}$$

5) Pooling: The pooling layer in neural networks combines pixels for smaller inputs. The two most common forms of pooling are MaxPooling and AveragePooling, in which the max and average functions are used to combine pixels, respectively. The pooling layer is typically applied after the convolutional layer, but before the activation function. This can further assist with better training [41]. For binary neural

network inference, it is also efficient to move the MaxPooling layer after the activation function; this does not require a change in logic, because it holds:

$$sign(max(x1, x2, x3...)) = max(sign(x1, x2, x3...)). \tag{8}$$

Since the sign function outputs a binary representation, the MaxPooling function can be represented as an OR gate [41].

6) Batch Norm: Batch normalization is essential to BNN training, and despite its complex nature, it reduces to an efficient operation during inference. The concept of batch normalization is to re-center the data around zero with a standard deviation of one. While the mean and standard deviation are updated during training, they are frozen during inference, allowing us to convert the data in an efficient way:

$$BN(x) = \frac{x - \mu}{\sqrt{\sigma^2 + e}} \cdot \gamma + \beta, \tag{9}$$

$$sign(BN(x)) = \begin{cases} +1 & \text{iff } (x > \tau), \\ -1 & \text{otherwise,} \end{cases} \tag{10}$$

where μ and σ^2 are the mean and variance of the batch, γ and β are the gain and bias, and e offers numerical stability. Moreover, $\tau = -\frac{\beta}{\gamma} \cdot \sqrt{\sigma^2 + e} + \mu$ and is calculated during weight preprocessing and rounded to the nearest integer [41].

7) Larq Library: The Larq library for binary neural networks is actively maintained, integrated into TensorFlow, and is well documented [42]. Its toolchain supports binary neural network training, and has many pre-trained models included as part of the platform. The API offers QuantDense and QuantConv2D layers for fully connected and 2D convolutions, respectively, in the BNN domain. In addition, it supports many sign activation functions, differing in their backward pass pseudo-gradient, as well as ternary [48] and DoReFa discrete activations [50].

E. Threat Model

The primary concern of this work is the privacy of both user data and proprietary network characteristics, such as biases. We assume an honest-but-curious cloud that will execute the correct operations on the encrypted data but is incentivized to snoop the uploaded data stored on their servers. Likewise, we consider cyberattacks that attempt to exfiltrate sensitive user data from the server. In addition, we assume that the user has limited knowledge about the network architecture and weights and is incentivized to learn proprietary secrets about the trained network.

In terms of user data, the cloud is able to determine the size and dimensions of the inference inputs. However, as encryptions of bits using the TFHE scheme are probabilistic and operations using encrypted ones and zeros take the same amount of time regardless of the underlying plaintext value, it is impossible for the cloud to deduce any information about the content of the user data.

With respect to the cloud's trained model, we acknowledge that by observing the magnitude of inference results for each class, it is possible to deduce the number of neurons in the second to last layer. For instance, if there are 200 neurons in the second to last layer, the maximum value of the scores for each class is around 200. However, the architecture of other layers and the model weights are protected from all curious users as it is impossible to glean any information about earlier layers from the magnitude of the output. We note that this threat can be thwarted by adding a batch norm or bias layer to the end of a network to blind this information.

III. Framework Overview

A. End to End System Overview

REDsec is an end to end framework that provides an efficient way to generate, train, and execute secure neural network inference. Our framework allows for configurable networks to be executed without writing complex blocks of FHE code. An overview of the REDsec modules is presented here, with references to Fig. 2. A detailed description of implementation details is provided in the next section.

- **REDsec Model Generator (1):** The model generator is a friendly UI tool used to generate a netlist of the model in CSV format. Use of this tool makes our system configurable and easy to use.
- **REDsec Compiler (2):** The REDsec compiler converts the CSV netlists into TensorFlow-based Larq training code and C++/CUDA secure inference net-file code.
- **Training with Larq (3):** Using the Larq library in Jupyter notebook, the model is trained on input data and the TensorFlow weights file is generated.
- **REDsec Library:** The C++/CUDA based REDsec library provides optimizations for efficient secure inference. This includes optimizations for network architecture, encrypted circuits and rigorous parallelism.
- **Weight Conversion with REDsec (4):** Utilizing the REDsec library and generated net-file code, the TensorFlow weight file is condensed and optimized to run using REDsec.
- **Secure Inference (Server) with REDsec (5):** Our server module is executed on a remote server. The C++/CUDA net-file code, which was generated by the REDsec compiler, utilizes the REDsec

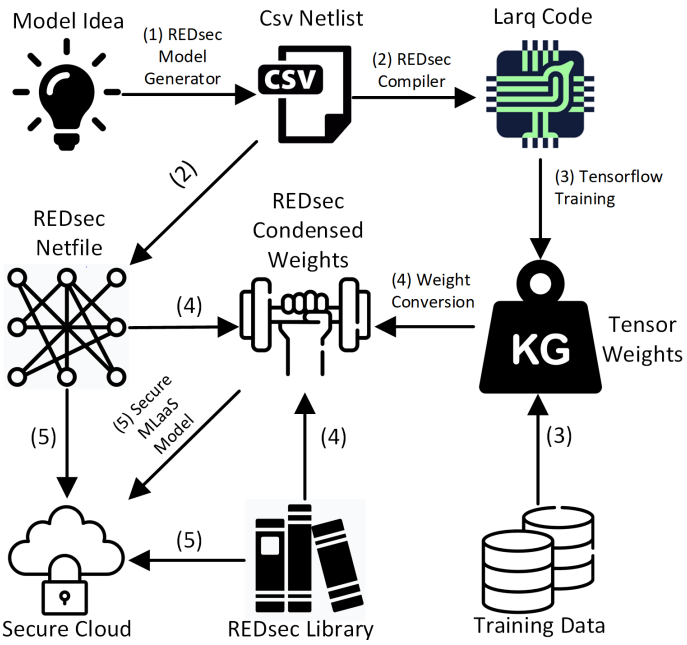


Fig. 2. **REDsec Overview:** Summary and interaction of the different components and modules of our REDsec framework.

library and reads the condensed weights file to run secure inference.

- **Secure Inference (Client):** Client modules are provided to prepare and encrypt the input data, send it to the secure inference server, and decrypt results.

B. Secure Inference Overview

REDsec is designed with cloud computing in mind: a remote user communicates with a cloud server, uploads inputs, and receives outputs in turn. REDsec includes client-side scripts that facilitate prepping private inputs, generating encryption keysets, and decrypting classification results. To enable private inference, the user and cloud will both need to initiate separate one-time setup phases. For the cloud, it must specify the neural network and train it. At this stage, the cloud service provider must provide a training set in the clear and a description of the neural network. The cloud will proceed to use the REDsec network compiler to generate code implementing the neural network using REDsec modules and will train the network in the plaintext domain with the training set in order to generate the weights. After the setup phase, the cloud is ready to receive private inference requests. For the user, a homomorphic keypair must be generated and the evaluation key sent to the cloud to facilitate homomorphic operations.

When the user wishes to classify a private input, she must first supply the data in either picture or binary form to a converter module, which will prepare the input and ensure that it is in the format that the network is expecting. This typically implies simply resizing or cropping the image, and centering the

pixels around the value of 0. For all networks in our experiments, we preprocess using:

$$preprocess(img) = 2 \cdot img - 255 \tag{11}$$

We note that this could have be implemented homomorphically on the cloud via a bitshift and integer subtraction. Next, the user will utilize an encryption script that will take the converted data, encrypt each bit using the private key generated in the setup phase, and export the resulting ciphertext array into a file. The user uploads this file to the cloud and then the cloud initiates the inference procedure.

The output will be in the form of a ciphertext array, the size of which depends on the network architecture and the number of possible classes in the dataset. For instance, the ImageNet dataset contains 1000 classes and the result of the inference will be 1000 scores of the input belonging to each class. Normally, these scores are converted using the softmax function, but to avoid executing costly floating point arithmetic circuits on encrypted data, we skip this analysis with REDsec. In the case of BinaryAlexNet for ImageNet, the outputs are fourteen bit unsigned integers, the highest of which indicates the class that the input most likely belongs to. After encrypted evaluation, the cloud will generate an output ciphertext file encoding the encrypted score of each class, and send this to the user.

Once receiving the output ciphertexts from the cloud, the user can use their secret key to decrypt the scores corresponding to each class and then simply take the maximum to find the correct classification. We opt not to do this in the encrypted domain as encrypted comparator circuits are not efficient (many gate evaluations required). In addition, the scores for each of the classes may provide relevant information to the user, especially in the case where the second-highest score is comparable to the maximum. Therefore, we leave it up to the user to sort or find the maximum of the dataset, depending on how they plan to use the data for their applications. This is the only computation involved on the user's behalf besides basic preprocessing, encryption, and decryption.

C. BNN Optimizations

1) Sign Function and Offset Conversion: The sign function in Equation 7 requires a comparison, which is expensive to perform in the encrypted domain. To get around this, we apply an offset value equal to $\text{sign}_{\text{offset}} = 2^{M_{bits}} - M/2$, where M_{bits} is the number of bits required to represent M, so that $M_{bits} = int(log2(M) + 1)$. We end up with the following expression:

$$bitsign(x) = \begin{cases} +1 & \text{iff } (x + offset) > 2^{M_{bits}}, \\ 0 & \text{otherwise,} \end{cases} \tag{12}$$

which can be simplified by taking the most significant bit of x. Since TFHE operates on bits, and assuming the ciphertext is encoded in the binary domain, the bit extraction is a free operation.

Furthermore, we need to add in multiple offsets throughout the layer. These include:

- 2's complement offsets in integer convolution (eq. 2),
- Ternary zero valued weight offset (eq. 5),
- Convolution bias offset (not typically used),
- Batch Norm offset (eq. 10), and
- Bit sign addition offset (eq. 12).

All of these offset values can be combined after training to condense the size of the weights file. Combining these offsets also means that during inference, each layer needs to apply only one M_{bit} addition per value. Therefore, even though REDsec's implementation of the activation function requires an addition, there is no additional cost to our activation function since it is combined with other operations.

2) Data Reuse: One useful concept in binary neural networks is the limited values that weights can take on. With this realization, we can perform basic operations only once and reuse that result.

Fully Connected and Convolution Layers: For the fully connected layer, the XNOR multiplications only need to be performed twice per value: once for a $\{$-1$\}$ weight, and once for a $\{$+1$\}$ weight. This observation simplifies the expensive and noisy bivariate XNOR gate to a low noise univariate NOT gate for $\{$-1$\}$ and a free copy operation for $\{$+1$\}$ (Fig. 1(b)). These results are stored in an array, and for every output filter applied, the corresponding output value of $\{$-1, +1$\}$ can be selected. Notably, this applies to both bitwise and integer multiplication. We also observe that $\{0\}$ values are independent of input values and are incorporated into the offset preprocessing. In practice, we combine the NOT operation with a bootstrap to convert from binary to integer ciphertexts.

3) Pooling Functions:

Average Pooling: In binary neural networks, average pooling may be used. Sum pooling has often been a substitute for leveled BGV based schemes, since homomorphic division is only possible in specific circumstances. To compensate for the change of function, BGV schemes adjust the weights in the subsequent convolution accordingly. We apply a similar approach here, but due to the nature of the sign function, there is no adjustments needed for the weights. However, we do need to sum the offset values so that they scale accordingly:

$$sign(avg(x1, x2, x3...)) = sign(sum(x1, x2, x3...)).$$

Max Pooling: Max pooling is not typically used in leveled BGV neural networks since a costly com-

parison must be made between values. For REDsec, we can use the method many BNNs use by moving the max pooling step to after the activation function. In this case, the max pooling problem is reduced to an OR gate:

$$sign(max(x1, x2, x3...)) = max(sign(x1, x2, x3...)).$$

D. FHE Optimizations

By default, both TFHE and cuFHE set parameters well suited for fast bootstrapping. In the standard, open-source implementations of both, it is assumed that bootstrapping will be done during every gate evaluation (with the exception of the homomorphic NOT gate, which essentially results in minimal noise growth). This paradigm is typically referred to as *gate bootstrapping* mode. This mode results in relatively slow homomorphic operations even with the superior bootstrapping capabilities of this scheme, on the order of 10-13 milliseconds for TFHE and 0.5 milliseconds for cuFHE. While this is indeed an impressive result in the field of FHE, it is still prohibitively slow for complex algorithms. For large neural networks, billions of gate evaluations are required to compute inferences; even small networks require millions of gate evaluations.

Efficient Operations: To compensate for this, we adopt two approaches, namely *bridging* and *lazy bootstrapping*. The first technique involves switching from the binary to the integer domain in order to evaluate efficient addition operations; this can be done with a single bootstrapping operation and is simply a matter of *dividing the TFHE torus into more segments*. With bridging we can minimize both noise growth (a single addition operation is relatively inexpensive in terms of noise accumulation) and eliminate the high number of bootstraps required to evaluate an addition circuit in the binary domain. The second approach involves choosing parameter sets that allow for a large number of leveled HE operations before bootstrapping is strictly necessary and monitoring the variance of noise levels of the output of each leveled HE operation. Since it is well known how additions and multiplications affect noise magnitude [36], and all TFHE operations (in both the binary and integer domain) are composed of additions and multiplications (modulo 2 for the binary case), we can accurately estimate the new noise variance after every type of computation on encrypted bits and integers. When the noise level exceeds a certain threshold, we perform a bootstrapping operation to allow us to execute another series of operations without bootstrapping.

Noise Auto-tuning: We also note that the TFHE bootstrap is integral to the evaluation of gate operations as it serves to scale the output to the correct region of the torus. In practice, no more than a few gates (depending on the gate types) can be evaluated on a ciphertext before bootstrapping is required to rescale

the underlying plaintext value. To avoid computing new noise variances for each ciphertext after each operation during actual inference, we designed a custom *auto-tuning* mechanism that only needs to be run once per network architecture. This procedure can either occur during the first live inference computation or on dummy data. This mode adds noise checks after each operation and designates locations in the network code where the noise variance exceeds a certain threshold. After all bootstrapping points are determined, there is no need to perform noise checks on subsequent inference computations since the noise will grow the same way each time. All arithmetic procedures constructed for convolutional, fully-connected, and pooling layers (as elaborated in Section IV) are optimized for low noise growth.

Parallelization: Further, due to the embarrassingly parallel nature of most neural network operations, we exploit multiple techniques to achieve the fastest performance for a given hardware target. If the system consists of strictly CPUs, we execute HE circuits in parallel with as many cores as available. Second, if GPUs are available, we execute cuFHE circuits in parallel by assigning each GPU streaming multiprocessor a separate circuit to execute. In this way, we can achieve inference speedups that scale linearly with the number of CPU cores and GPUs.

(RED)cuFHE: Lastly, we introduce major overhaul to the cuFHE library for efficient GPU evaluation of homomorphic circuits. First, the original cuFHE library only supports a single set of parameters corresponding to the recommendations set forth in the original TFHE paper [24]. This parameter set corresponds to 110 bits of security and while it is a solid default configuration, it is not optimal for all types of applications and algorithms. We introduce changes to the library to allow for customizing injected noise levels and degrees of LWE polynomials (which can be used to influence sizes of both keys and ciphertexts) to allow finding optimal parameter sets. In addition, this allows users to tweak the level of security to their desired setting.

Moreover, we introduce *leveled gates* (i.e., gates without any bootstrapping) and a robust API for leveled integer addition and scalar multiplication to the library as well as encrypting constant values. These constant values are considered public and are encoded by the cloud with zero noise. The impact of this mechanism is to allow private information from the user *to be mixed with non-private data*. In the case of neural network inference, values such as biases are converted into noiseless, encrypted constants in order to interface with the uploaded inputs encrypted with the secret key.

IV. Implementation Details

A. Model Generation and Training

In order to make our REDsec framework more accessible, we developed a bespoke compiler to encrypted neural networks. The input to the compiler is a CSV netlist outlining each of the neural network layers, while the REDsec model generator was developed using Excel VBA to generate this netlist. The tool directly asks for the convolution dimensions, pooling options and batch normalization requirements of the desired network, and it further provides room for dropout to be applied. After the REDsec model generator outputs the CSV netlist, the REDsec compiler can be used on the netlist to generate the training and secure inference code.

For training, the source code output is a Jupyter notebook file that leverages TensorFlow and the Larq library. This notebook file can be executed locally using Jupyter, or can run on cloud hosts (e.g., one can use Google Colab for debugging or rapid prototyping). After the correct training and validation dataset files are uploaded and properly linked, the code can be executed to train the network. As soon as the network is trained in TensorFlow, a final weight extraction and compression must be performed. The REDsec secure inference code comes with an integrated weight converter that transforms TensorFlow's floating point weights to ternary weights, and also combine different offsets, as discussed in Section III-C1. Finally, the weight converter outputs a compressed weight file that is used for secure inference.

B. Secure BNN Inference with REDsec

The inference code leverages the REDsec, TFHE, and modified (RED)cuFHE libraries and consists of both C++ and CUDA modules depending on available hardware. These files are run on the cloud to perform secure inference.

1) Server Modules: After invoking the compiler in the training step, the cloud service provider should already have a compressed weights file and the high level net code that is integrated with the REDsec library. Once the server receives the evaluation key of a client, it can run noiseless encryption scripts to compute biases for integration with secure inputs uploaded by users. Once a user sends an inference request and encrypted input data to be classified, the cloud can execute the generated net code and send the encrypted result back to the user after the inference procedure is complete. As it is possible to encode multiple bits in a single ciphertext, the generated net code takes advantage of this to minimize communication overhead and memory consumption by *packing results of classification into a single ciphertext per class*.

2) Client Modules: There are three primary programs that are run on the client-side, none of which are computationally demanding. The first is a key generation script that allows users to specify a security level (in bits) and creates a keypair using either TFHE or (RED)cuFHE depending on whether the user wants the cloud to use CPU cores or GPUs for homomorphic computations. The user is expected to store the generated secret key, which is needed by the other client scripts for encryption/decryption and uploads the evaluation key directly to the cloud server.

After generating a secure keypair, the user can utilize the second client module to prepare any inputs for secure inference. First, the raw bytes of the image are read and checks are performed to ensure that the image is compliant with the network architecture. REDsec will ensure that the image dimensions are appropriate for the network and determines how many color channels to use; depending on the network, this is either one for black and white images or three for color images. Next, the values of each pixel are read from left to right and top to bottom and each color channel is encrypted as an array of eight ciphertexts in the binary domain (since the value can vary from 0-255) and appended to a large ciphertext file that holds all of the encrypted bits of the image. To start the outsourced secure inference, the user uploads this ciphertext file to the cloud that evaluates the classification result.

The cloud will return a single ciphertext file storing the results of the neural network inference. This file, like the input ciphertext file, will vary in size depending on the application and network architecture itself. The ciphertext result file will consist of a number of ciphertexts equal to the number of possible classes in which the input can belong. The magnitude of each ciphertext (which is an integer) can grow up to the number of neurons in the second to last layer of the network, specifically $\lceil \log_2 N \rceil$ where N is the number of neurons. The decryption module will use the secret key to process each ciphertext corresponding to each class and return the plaintext scores. The class with the highest score indicates the most likely match with the input image and can be determined by simply computing the max of the values for all classes.

C. REDsec Library Implementation

The REDsec library contains our implementations of the TFHE machine learning circuits. For additional flexibility, it can be compiled in unencrypted mode for debugging, or encrypted mode for evaluation. This section gives an overview of the library and the optimizations that were contributed.

1) Library Structure: The REDsec library contains the following layers of abstraction:

- **Layer:** The layer library encapsulates convolution, pooling, batch normalization and activation into a single object. The layer object ensures proper order of these functions so that the REDsec

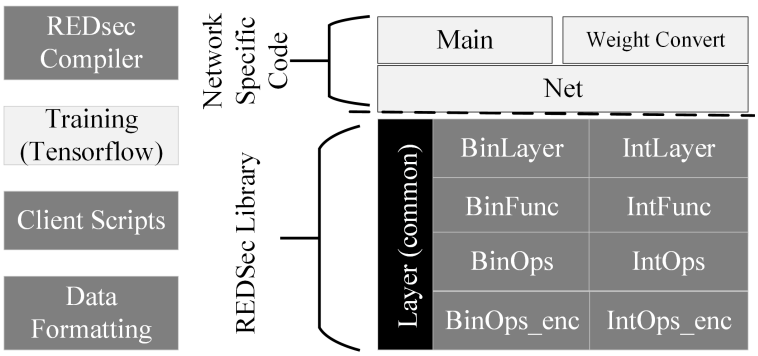


Fig. 3. **REDsec C++/Cuda Library:** This figure shows the modules in the REDsec library, as described in Section IV-C1.

optimizations are preserved.

- **Func:** This level of files contain optimized implementations of convolution, fully connected, pooling, batch normalization and quantize activation functions. OpenMP-based parallelization is added at this level of abstraction.
- **Ops:** The Ops files contain basic, low level logic and arithmetic circuits that invoke the underlying cryptographic library directly. This part of the library is where the encrypted operations are implemented. Depending on the hardware target, the encrypted operations are written in CUDA (for GPUs) and C++ for CPU-based systems.

In addition to these levels of abstraction, we subdivide the functions into integer and binary components based on the layer input. Therefore, the user can decide to use integer layers for higher accuracy or binary layers for speed.

2) Encrypted Circuit Designs: The goal of the homomorphic circuits utilized for neural network inference is to minimize the noise growth in the ciphertexts in order to delay bootstrapping as long as possible. The goal is to only bootstrap when a conversion from the integer domain to the binary domain and vice-versa is required, as this will serve to accomplish the conversion and noise reduction at the same time. Since bootstrapping remains the bottleneck of FHE operations and takes up significantly more time than a standard ciphertext addition in TFHE, we aim to perform this procedure as few times as necessary, yet guarantee correct decryption result for the client. In this subsection, we present an outline of the core building blocks of homomorphic inference that are used to construct the different network layers.

Adder constructions: We observe in neural network inference that adder circuits operating on bits form the most computationally expensive operations in evaluating REDsec networks. In our adder designs for

unsigned and signed adders, the basic building block is a full adder using using two XOR gates, two AND gates, and an OR gate. In total, this requires 10 arithmetic operations on LWE ciphertexts to evaluate [24]. The primary reason for the large cost of using binary adder circuits is the bootstrapping required to successfully evaluate it.

We made a key observation with regards to noise accumulation in the propagating carry bits used in a ripple chain for multi-bit adders. We observe that the carry accumulates far more noise than any other ciphertext involved in the circuit. This is because it is the only ciphertext object involved in every stage of the adder and is continuously computed upon to determine the carry for the next stage. In addition, it mixes at each stage with the secure input ciphertexts, resulting in an exponential noise growth in the most significant bits of the result. In a carry-ripple design, even rigorously optimizing for noise accumulation, the carry bit needs to be bootstrapped frequently in the carry-ripple chain. We note that parallel adders can reduce the size of the carry chain, but this only serves to marginally improve the problem, requiring slightly fewer bootstraps.

Instead, using the special properties of the TFHE bootstrap, *we can rescale the ciphertexts* from the binary message space to an integer space (modulo an integer representing the total number of regions on the TFHE torus). This will allow us to use the natural FHE addition operation instead of a costly adder circuit composed of logic gates to compute the sum of two ciphertexts. Instead of dozens of bootstraps and even more ciphertext arithmetic operations, we can accomplish this procedure for the cost of a single bootstrap for the conversion plus the negligible cost of a single ciphertext addition.

Multiplication: These circuits are among the slowest to execute of the basic arithmetic circuits using TFHE. However, because REDsec constructs BNNs and not full precision networks, the multiplication operation is simply a single NOT operation. In the TFHE cryptosystem, this NOT operation does not require a bootstrap procedure and becomes among the fastest operations in REDsec networks.

Activation: The most efficient operation in REDsec is the computation of the *sign function*, which is used as the activation function for REDsec networks. While other works that utilize the sign function need to extract the top bit of a ciphertext representing integers, which is an expensive and complicated operation, or perform a programmable bootstrap, we need only make a copy of the ciphertext representing the MSB of the ciphertext vector, assuming the current encrypted value is currently in the binary domain. This operation is fast, accumulates no noise, and is essentially free, which is a major motivation for using BNNs and treating ciphertexts as individual bits for certain operations in the first place.

3) GPU Modules for Encrypted Computation: GPUs can be used to achieve significant speedups over strictly CPU based systems for homomorphic operations, particularly bootstrapping. For example, a GPU

can achieve over a 37x speedup compared to a CPU for bootstrapping operations using the TFHE scheme [38]. For this reason and the fact that cloud instances with GPUs are widely available (such as the P and G families of Amazon EC2 instances), the REDsec library provides GPU support for all homomorphic operations through the use of CUDA code, the base cuFHE library, and several REDsec optimizations built on top of cuFHE. Also, through the use of a custom GPU scheduler, REDsec is able to effectively utilize GPU resources for any arbitrary number of available GPUs.

Updates in (RED)cuFHE: Like the TFHE library, cuFHE only exposes bootstrapped gate functions to users. Even though the bootstrapped operations are much faster on GPUs, they are still orders of magnitude slower than their leveled equivalents. As such, we constructed leveled gate functions and arithmetic operations to fit with our lazy bootstrapping paradigm and modified the cuFHE ciphertext structures to add variables that track of current noise variance. This variable, similarly to the TFHE library, accumulates differently for various types of operations involving encrypted data. This allows REDsec to utilize our *auto-tuning* feature to predict the best places in the network to insert bootstrapping operations. Further, we parameterize cuFHE to allow for different configurations corresponding to various security levels, as determined using the LWE estimator framework [51].

GPU resource scheduler: In order to maximize resource utilization for any number of GPUs, we incorporate a custom scheduler to assign GPU streams to CPU threads running inference operations. The scheduler runs on a dedicated CPU thread that maintains an array to keep track of resource utilization and uses shared memory to direct CPU threads to specific GPUs and stream handles. When a CPU thread needs to outsource FHE computation, it will ask the scheduler for a number of GPU streams proportional to the work that needs to be done by entering a shared, thread-safe queue and the scheduler will return new stream handles as they become available and when it is at the front of the queue. In the meantime, the CPU thread can utilize its currently assigned hardware resources while it waits for more assignments. When possible, the scheduler will attempt to find resources on a single GPU for any given CPU thread, as communicating with multiple GPUs on a single thread will result in unnecessary overheads. For instance, the CPU thread will need to transfer data back and forth between the GPUs it is using and also switch contexts in CUDA (since a single GPU can be active at any moment on a single thread) which impacts performance.

V. Experimental Evaluation

To verify and test the efficiency of our framework, we conduct experiments using several network architectures to classify images from three popular datasets at various levels of complexity. For experi-

Fig. 4. **ImageNet:** This graphic, taken from the 2015 ImageNet paper [53], shows a comparison between the simple classes in PASCAL or CIFAR-10 and the much more nuanced classes of the Imagenet dataset.

ments with a GPU-based backend, we employ a g4dn.metal AWS instance containing eight NVIDIA T4 GPUs, which contain 40 streaming multiprocessors each, and 96 vCPU cores running at 2.5 GHz. To run experiments purely on CPUs, we utilize a c5d.24xlarge AWS instance with 96 vCPU cores running at 3 GHz. We configure TFHE for 80 bits of security and cuFHE for 110 bits of security.

A. Dataset and Model information

We employ the following datasets for evaluation:

MNIST: The Modified National Institute of Standards and Technology (MNIST) is a dataset of *28x28x1* images of handwritten digits, ranging from 0 to 9, resulting in 10 output classes. Since the input is composed of simple black and white pixels, a very small neural network can evaluate this dataset effectively [52].

CIFAR-10: The CIFAR-10 dataset has small *32x32x3* images that represent 10 different basic classes. This dataset is much more complicated than MNIST, due to the multicolor input channels, curved lines and different types of inputs. Example classes include birds, cats and dogs.

ImageNet: The Image dataset has large images of various sizes and has 1000 classes of similar objects. For example, while CIFAR-10 had one class for all birds, Imagenet has 59 different classes for birds. There are many different architectures to classify these images, with the most popular being the ResNet

TABLE II. **SECURE INFERENCE EXPERIMENTS:** THE FIVE NETWORKS DEPICTED WERE RAN USING 96 VCPUS (FOR CPU-EVAL) AND 8 GPUS (FOR GPU-EVAL)

	FF_{96}	$CCFF_{3296}$	$BNet_S$	BNet	BAlexNet
DataSet	MNIST	MNIST	CIFAR-10	CIFAR-10	ImageNet
Input Size	28x28x1	28x28x1	32x32x3	32x32x3	224x224x3
Classes	10	10	10	10	1000
Layers	2	4	8	9	8
Int MACs	0	0	1.6M	3.1M	72.9M
Bin MACs	19.7k	1.27M	58.4M	511.0M	768.5M
Bin Weights	19.7k	317k	2.0M	10.4M	61.8M
Int Weights	106	170	1.7k	3.9k	11.4k
Ptxt Eval (s)	0.0014	0.058	2.77	28.3	159
CPU-Eval (s)	3.5	41.5	527	1753	10530
GPU-Eval (s)	0.78	17.4	373	1708	8720
MAC Cost (us)	39.5	13.7	6.2	3.4	10.4
Accuracy (%)	93.1	97.8	81.9	88.5	61.5 [1]

[1]Top-5 accuracy for the BinaryAlexNet architecture, as reported in [44].

and AlexNet architectures. We chose a pretrained BinaryAlexNet architecture to evaluate on REDsec, due to its sequential nature and binary layers. Image preprocessing includes resizing the image to a *224x224x3* image, then centering the inputs around zero by using Equation (11).

B. Inference Results

The results of REDsec for five neural network architectures as well as network characteristics are shown in Table II. When timing the inference procedure, we do not include the time required for key I/O and memory allocation. Instead, we compute an amortized cost over five back-to-back inferences after this setup phase is complete. Specifically for GPU evaluation, this setup time can take up to a few minutes depending on the complexity of the network in order to allocate pinned memory regions on the host and memory on the devices to prepare for subsequent inferences. However, we remark that this is a one-time cost and the server can run an arbitrary number of inference procedures in a session without re-allocating memory.

MNIST: For our networks to evaluate MNIST, we use a network called FF_{96}, a two layer fully connected network with 96 intermediate neurons and 10 output neurons, and $CCFF_{3296}$, a network with two 3x3 convolutional layers of output depth 32, and two fully connected layers having 96 intermediate neurons

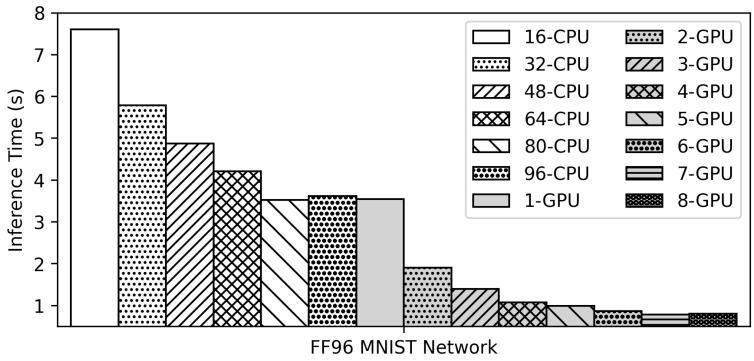


Fig. 5. **FF$_{96}$ Inference:** The FF$_{96}$ network is a small, two layer network consisting of solely fully-connected layers that is employed for classification with MNIST. With 7 GPUs, a single image can be classified in only 0.8 seconds.

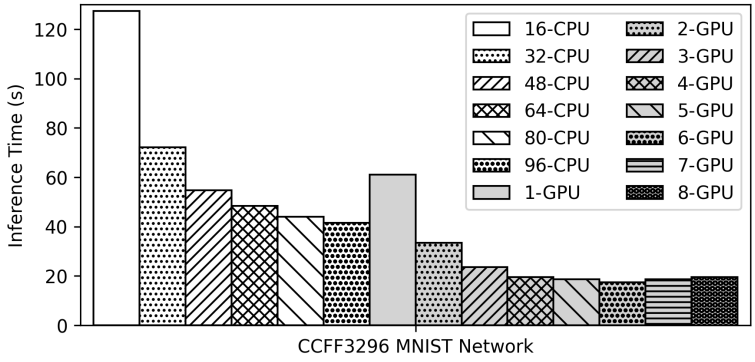


Fig. 6. **CCFF$_{3296}$ Inference:** This network boasts a higher accuracy for MNIST than FF$_{96}$ and contains two additional convolutional layers. The fastest configuration for this network is 6 GPUs, taking only 17.5 seconds per inference procedure.

and 10 output neurons. An encrypted 2x2 MaxPool was placed at the very beginning of the network to reduce the input size.

For the FF$_{96}$ network, we were able to achieve a 93.1% accuracy for the MNIST dataset. Fig. 5 shows our inference costs per image for a variety of hardware resources. The results demonstrate the speedups that can be achieved by REDsec when leveraging multiple GPUs and result in sub-second inference times for this network. The CCFF$_{3296}$ network offers an alternative solution to MNIST classification with higher accuracy, but also significantly more complexity. Two convolutional layers are added at the start of the network and comprise the majority of the execution time. Fig. 6 shows similar trends as the FF$_{96}$ network and emphasizes the inherent opportunities for parallelism in both convolution and fully connected layers.

CIFAR: We use two architectures to classify the more complex CIFAR-10 dataset, both based on a

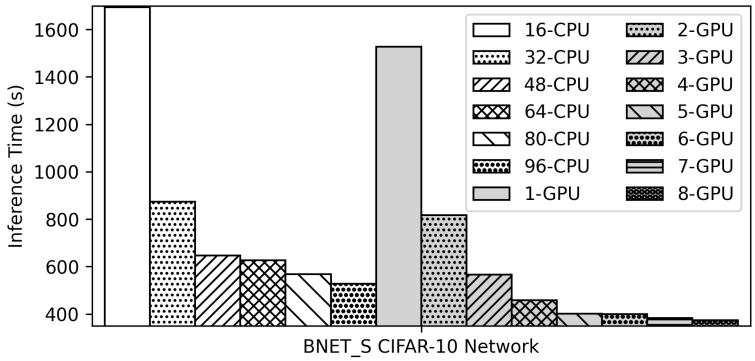


Fig. 7. **BNet$_S$ Inference:** a CIFAR-10 network with 5 convolutional layers and 3 fully connected layers. The fastest speeds achieved utilized 8 GPUs and completed evaluation in approximately 373 seconds.

CIFAR-10 BNN design from BinaryNet [39]. The BNet$_S$ network is a scaled down version of the original network design and was optimized for speed while maintaining acceptable levels of accuracy. The resulting network uses valid padding and half of the depth of the original BinaryNet architecture in [39]. Notably, this provides a good middle-ground benchmark between the relatively simple MNIST network designs and the far more complicated full BNet and AlexNet networks. Our results for this network are depicted in Fig. 7. We remark that the heightened inference times for 1 and 2 GPU configurations are primarily due to memory transfer costs as the input images are larger and the memory required to store intermediate ciphertext values is much greater than in either MNIST networks. We observe that this is the case for all large networks that we study in this work. Most of this overhead is mitigated through the use of asynchronous transfers that overlap with kernel executions, yet the overall performance is still impacted in 1-2 GPU configurations.

ImageNet: Despite the memory transfer costs just outlined for GPU evaluation, the binary AlexNet evaluation GPU implementation still outperforms the CPU implementation by approximately 30 minutes (Table II). In terms of the cost of MACs (Multiply-Accumulates), as the networks get larger, we observe a higher throughput for these operations. This can be attributed to the better resource utilization resulting from the presence of more work in these bigger networks. Overall, REDsec performs well for a variety of different network architectures of varying depths and complexity.

VI. Related Works

Secure inference has three different branches of solutions: strictly LHE, multi-party computation (sometimes combined with PHE or LHE) and FHE solutions. LHE solutions include CryptoNets [12], Faster CryptoNets [54], n-Graph HE [55] and CryptoDL [56]. These schemes use polynomial activation

functions, including the square function x^2 [12], trained quadratic functions $a \cdot x^2 + b \cdot x$ [55] and n-degree polynomials for ReLU, tanh and sigmoid approximations [56]. These approximations either suffer from accuracy loss or generate a lot of noise. The polynomial activation functions are also not typically used by plaintext neural networks, so they lack library support. In addition, LHE schemes also use a technique called *batching* to improve data throughput. This technique allows users to upload multiple images at a time. However, despite high throughput, the latency is slow. Another downside is that, in order to maximize the number of slots that can be used in batching, one will sacrifice any hope of a fast bootstrapping procedure. This is a key reason why batching does not play a large role in FHE evaluation. REDsec, on the other hand, is optimized for low latency through its use of the TFHE library. Most users will not be able to take advantage of the high number of slots that batching provides (typically up to 8192 slots), as this would require an individual user to need thousands of images to be classified at once.

In addition, binary neural networks are well supported for low-memory edge computing devices [40], [41]. Naturally, these resource-constrained devices are good candidates for secure MLaaS, and those using binary neural networks can easily be upgraded to use the REDsec MLaaS system. Finally, LHE schemes are not scalable to larger networks due to noise accumulation forcing incorporation of increasingly less efficient parameter sets. One can use bootstrapping to make these schemes (e.g., BGV and CKKS) into FHE constructions, but preliminary results in this area have yielded prohibitively slow evaluation times. In the case of CKKS, it was demonstrated that inference using a ResNet-20 model on the CIFAR-10 dataset takes approximately four hours [57]. REDsec, on the other hand, uses TFHE-based schemes, which have efficient bootstrapping methods and allow for neural network architectures with unlimited depth.

One leveled homomorphic network, SHE, differs from the other networks in that it uses TFHE [62]. Since bootstrapping is not considered in their design, the network still suffers from the scalability problems mentioned for other LHE frameworks. SHE attempts to implement a full precision neural network by approximating weights to a power of two. This multiplication optimization allows the circuit to be an XNOR followed by a bitshift [62]. While this optimization is sometimes used with mixed precision BNNs, it is not yet supported in the current version of the Larq library [42], [62], [41], [64], [50]. While SHE supports a ReLU function, this activation is not commonly used in pure binary neural networks [41]. The main limitation of SHE is that its architecture is only built upon the boolean logic gate operation in TFHE, which is very slow. This results in increased inference times compered to using integer arithmetic [62], [65]. While the full AlexNet is not implemented in SHE, its authors indicate that their implementation would take 24.7 hours to run [62], compared to our experimental evaluation of AlexNet that runs in 2.4

TABLE III. **COMPARISON OF COMMON NN FEATURES:** SUMMARY OF NN FEATURES AVAILABLE INTO THE PRIVACY-PRESERVING DOMAIN.

Features	FHE	Dataset Size	Weight Size	Fully Conv.	Fully Connected	MaxPool	AvgPool	Batch Norm	Dropout	Standard Activation	Int. Inputs	Configurable	Implemented
Cheetah [18]	○	●	○	●	●	◐[1]	○	○	○	◐[1]	●	○	●
Gazelle [17]	○	◐	○	●	●	◐[1]	○	○	○	◐[1]	●	○	●
SecureML [58]	○	○	○	○	●	○	○	○	○	◐[1]	●	○	●
XONN [59]	○	◐	○	◐[1]	◐[1]	◐[1]	○	●	○	●	●	○	●
MP2ML [60]	○	○	○	●	●	◐[1]	●	○	●	◐[1]	●	○	●
MiniONN [19]	○	◐	○	◐[1]	◐[1]	◐[1]	◐[1]	●	●	◐[1]	●	●	●
CryptoNets [12]	○	○	○	●	●	○	●	○	○	○	●	○	●
Faster CryptoNets [54]	○	○	○	●	●	○	●	●	○	◐[2]	●	○	●
n-GraphHE [55]	○	●	○	●	●	○	●	●	●	●	●	●	●
CryptoDL [56]	○	◐	○	●	●	○	●	●	●	◐[2]	●	●	●
Concrete [15]	●	○	○	●	●	●	●	●	○	●	●	○	●
FHE-Dinn [14]	●	○	●	○	●	○	○	○	○	●	○	○	●
Hopfield [61]	●	○	●	○	○	●	○	○	○	●	○	○	●
SHE [62]	●	◐	○	●	●	●	○	●	○	●	●	○	○
Tapas [63]	●	○	○	●	●	○	○	○	○	●	○	○	○
REDsec	●	●	●	●	●	●	●	●	●	●	●	●	●

[1]These operations are done by the client using garbled circuits.

[2]These operations are polynomial approximations.

hours. Currently, the available implementation of SHE covers only basic functional units, instead of a full secure inference neural network [66].

The second class of secure inference solutions involve multi-party computation [18], [17], [58], [59], [60], [19]. These schemes send the data back to the user after every layer, and require them to perform the non-linear activation function. One benefit of this model is the ability to apply normal activation functions, such as ReLU or sigmoid. However, this imposes a significant computational overhead on the user, which diminishes one of the motivations for MLaaS: outsourced, offline computation. It also incurs communication overheads of tens to hundreds of megabytes [17], [59]. REDsec allows for computations to be done entirely on the cloud. In addition, the sign activation function is the function of choice for binary neural networks, which our framework evaluates very efficiently.

One comparable MPC solution is XONN, which also uses BNNs to perform secure inference [59]. XONN proposes using the garbled circuit protocol to speed up additions and perform the comparison

TABLE IV. **RESULT COMPARISON:** MAX MACS CORRESPONDS TO THE BIGGEST KNOWN MODEL EVALUATED ON THE FRAMEWORK IN TERMS OF MILLIONS OF MULTIPLY-ACCUMULATE (MAC) OPERATIONS. REDSEC INFERENCE SCALES TO NETWORKS NEARLY 10X LARGER THAN THE CONCRETE FRAMEWORK AND ORDERS OF MAGNITUDE LARGER THAN FHE-DINN, HOPFIELD AND XONN.

Performance	REDsec (this work)	Concrete [15]	FHE-Dinn [14]	Hopfield [67]	SHE [62]	XONN [59]
Biggest Dataset	ImageNet	MNIST	MNIST	Custom[1]	ImageNet[2]	MNIST
Solution	FHE	FHE	FHE	FHE	LHE	MPC
Max MACs (M)	841.3	9.1	0.02	0.2	28	20.2
Weight Size	Micro	X-Large	Small	Small	Small	Large
Training Lib.	TensorFlow	N/A	Custom	Custom	N/A	Custom
Input Type	Int	Int	Bin	Bin	Int	Bin
Weight Type	Tern	Int	Int	Scaled	Scaled	Scaled
Activ. Type	Bin	Int	Bin	Bin	Int	Bin
MNIST						
Train Time	5 minutes	N/A[3]	11 hours	N/A[3]	N/A[3]	N/A[3]
Inference (ms)	781	1800	1640	N/A[4]	9300	1750
Accuracy (%)	93.1	97.1	96.4	N/A[4]	99.54	97.6
MAC Speed (us)	3.4	8	80	3	642	17

[1]An custom Facial Recognition dataset, which is no longer available, was used. Heavy feature extraction was performed on the dataset before feeding it into the network.

[2]ShuffleNet[68], an architecture one order of magnitude smaller than the AlexNet[43] architecture used in this paper, was used for evaluation.

[3]These works have no open source code to evaluate the network.

[4]These works did not implement MNIST.

in the sign function. However, the limitations of communication overhead and involving the user in the calculation are still present for XONN. Like SHE, XONN uses a power of two scaling factor, is not currently accessible as open source, and does not have a supporting TensorFlow framework to train the network [59], [60].

The third class is the fully homomorphic branch of solutions, to which REDsec belongs. There are few existing solutions today that use this approach, but since the introduction of TFHE with its efficient bootstrapping procedure, several works have proposed unique solutions, as summarized in Tables III and IV. FHE-DiNN [14] uses binary neural networks to achieve inference, and its main innovation is the programmable bootstrapping operation, where a lookup table is used during a bootstrap instead of performing an activation function. While this allows for generic non-linear activations, the sign function

can be calculated very efficiently for our scheme in the binary representation. In addition, REDsec optimizes convolution, max pooling, sum pooling and leveled operations through the use of bridging, all of which are not supported in FHE-DiNN. Finally, FHE-DiNN is hard to train, not configurable, and does not rely on any current library for training [14], [69]. This makes it difficult to use in practice.

Another contemporary work called Concrete focuses on developing integer based functions with programmable bootstrapping [15]. Their sample neural network codebase is not published, making it difficult to compare to our work [70]. Nevertheless, based on the description, their biggest network has around one million multiply-accumulate (MAC) operations (92 inputs$\cdot 92$ outputs$\cdot 100$ layers) [15]. As a comparison, our AlexNet has 841 million MAC operations. Concrete also uses encrypted integer weights, with each TFHE encrypted weight on the order of kilobytes in size, compared to REDsec that can use 2-bit unencrypted weights due to the unique features of ternary neural networks. This results in a size compression of multiple orders of magnitude and allows REDsec to scale to AlexNet, which has 61.8 million weights. Finally, our max operation for MaxPooling does not require a bootstrap for common window sizes, unlike Concrete, and our convolution requires us to bootstrap the inputs once, as opposed to Concrete which requires two bootstraps for every multiplication operation [15].

A third work, TAPAS, presents theoretical approaches to secure inference [63]. While no neural network was constructed, timing examples in the paper are extrapolated from evaluation on smaller functional units. In addition, implementation details such as ternary networks, data reuse, integer activations, and bridging were not explored in this paper. Finally, since no material neural network was produced, it remains difficult for future works to expand on their approach [63].

Finally, a hopfield network architecture was recently proposed for secure inference [61]. A hopfield network is a binary recurrent neural network of one or up to a few layers. It was first proposed in 1982 as one of the first machine learning networks, and some recent applications have made use of them because neurons can run independently in parallel [67], [71]. Given its binary nature, this network architecture is a good choice for secure inference, but it is not as widely used as conventional BNNs for most applications [61], [41]. This solution also uses a small dataset of faces and requires heavy unencrypted preprocessing and feature extraction of facial images by the user. We observe that in terms of multiply-accumulate (MAC) operations, this network is several thousand times smaller than the BinaryAlexNet evaluated in our work.

VII. Conclusion

We have presented REDsec, an end-to-end framework for binary neural network training and secure inference using fully homomorphic encryption. As part of the framework, we developed a compiler to output both TensorFlow training and C++/CUDA secure inference code for easy adoption by data scientists and researchers. The secure inference framework takes advantage of binary and ternary network operations to select FHE-friendly functions for convolution, fully connected, max pooling, average pooling and quantization activation layers. Furthermore, we upgraded the cuFHE framework by instituting encryption of constants, leveled operations and bridging between binary and integer ciphertexts in order to maximize addition throughput. We demonstrate the capability of our framework by evaluating MNIST, CIFAR-10 and ImageNet, with encrypted speeds only 1.7 to 2.7 orders of magnitude slower than an un-vectorized plaintext CPU implementation.

A future research direction involves including extensions for homomorphic ReLU and other nonlinear activation functions. This would allow for encrypted evaluation of mixed precision networks, which have higher accuracy than pure BNNs. We will also investigate adding support for residual networks, like ResNet, to our framework, to allow for different styles of DNNs and determine the feasibility of FHE for these networks.

References

[1] P. Papadopoulos, N. Kourtellis, P. R. Rodriguez, and N. Laoutaris, "If you are not paying for it, you are the product: How much do advertisers pay to reach you?" in *Proceedings of the 2017 Internet Measurement Conference*, 2017, pp. 142–156.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 199–212.

[3] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 305–316.

[4] R. Miller, "Foiled plot to attack amazon reflects changing nature of data center threats," Apr 2021. [Online]. Available: https://datacenterfrontier.com/foiled-plot-to-attack-amazon-reflects-changing-nature-of-data-center-threats/

[5] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 896–902.

[6] A. P. Tafti, E. LaRose, J. C. Badger, R. Kleiman, and P. Peissig, "Machine learning-as-a-service and its application to medical informatics," in *International Conference on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2017, pp. 206–219.

[7] Centers for Medicare & Medicaid Services, "The Health Insurance Portability and Accountability Act of 1996 (HIPAA)," Online at http://www.cms.hhs.gov/hipaa/, 1996.

[8] NIST, "Advanced Encryption Standard (AES)," in *FIPS PUB 197, Federal Information Processing Standards Publication*, 2001.

[9] A. Sachdev and M. Bhansali, "Enhancing cloud computing security using aes algorithm," *International Journal of Computer Applications*, vol. 67, no. 9, 2013.

[10] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[11] R. Cramer, I. B. Damgård *et al.*, *Secure multiparty computation*. Cambridge University Press, 2015.

[12] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International Conference on Machine Learning*. PMLR, 2016, pp. 201–210.

[13] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 142–156.

[14] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Annual International Cryptology Conference*. Springer, 2018, pp. 483–512.

[15] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," Cryptology ePrint Archive, Report 2021/091, 2021., Tech. Rep., 2020.

[16] D. Mouris, N. G. Tsoutsos, and M. Maniatakos, "Terminator suite: Benchmarking privacy-preserving architectures," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 122–125, 2018.

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

[18] B. Reagen, W. Choi, Y. Ko, V. Lee, G.-Y. Wei, H.-H. S. Lee, and D. Brooks, "Cheetah: Optimizations and methods for privacy preserving inference via homomorphic encryption," *arXiv preprint arXiv:2006.00505*, 2020.

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

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

[21] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *International Workshop on Public Key Cryptography*. Springer, 2012, pp. 1–16.

[22] S. Halevi and V. Shoup, "Bootstrapping for helib," in *Annual International conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 641–670.

[23] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.

[24] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *international conference on the theory and application of cryptology and information security*. Springer, 2016, pp. 3–33.

[25] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: Accelerating Fully Homomorphic Encryption

by Bridging Modular and Bit-Level Arithmetic," Cryptology ePrint Archive, Report 2018/1013 v.20200219, 2020, https://eprint.iacr.org/2018/1013/20200219:192239.

[26] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[27] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.

[28] A. Blum, A. Kalai, and H. Wasserman, "Noise-tolerant learning, the parity problem, and the statistical query model," *Journal of the ACM (JACM)*, vol. 50, no. 4, pp. 506–519, 2003.

[29] S. Khot, "Hardness of approximating the shortest vector problem in lattices," *Journal of the ACM (JACM)*, vol. 52, no. 5, pp. 789–808, 2005.

[30] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[31] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[32] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[33] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

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

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

[36] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*. Springer, 2013, pp. 75–92.

[37] W. Dai and B. Sunar, "cufhe (v1.0)," 2018. [Online]. Available: https://github.com/vernamlab/cuFHE

[38] NuCypher, "nufhe (v0.0.3)," 2019. [Online]. Available: https://github.com/nucypher/nufhe

[39] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[40] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[41] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6, p. 661, 2019.

[42] L. Geiger and P. Team, "Larq: An open-source library for training binarized neural networks," *Journal of Open Source Software*, vol. 5, no. 45, p. 1746, Jan. 2020. [Online]. Available: https://doi.org/10.21105/joss.01746

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

[44] Larq, "BinaryAlexNet on Larq Zoo Pretrained Models," https://docs.larq.dev/zoo/api/literature/#binaryalexnet.

[45] A. Prabhu, V. Batchu, R. Gajawada, S. A. Munagala, and A. Namboodiri, "Hybrid binary networks: optimizing for accuracy, efficiency and memory," in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2018, pp. 821–829.

[46] S. Zhu, X. Dong, and H. Su, "Binary ensemble neural network: More bits per network or more networks per bit?" in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4923–4932.

[47] I. Chakraborty, D. Roy, I. Garg, A. Ankit, and K. Roy, "Constructing energy-efficient mixed-precision neural networks through principal component analysis for edge intelligence," *Nature Machine Intelligence*, vol. 2, no. 1, pp. 43–55, 2020.

[48] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[49] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[50] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[51] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[52] Y. Lecun, L. Bottou, Y. Bengie, and P. Haffner, "Mnist dataset," https://www.tensorflow.org/datasets/catalog/mnist, 1994.

[53] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[54] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," *arXiv preprint arXiv:1811.09953*, 2018.

[55] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "ngraph-he2: A high-throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019, pp. 45–56.

[56] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *arXiv preprint arXiv:1711.05189*, 2017.

[57] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *arXiv preprint arXiv:2106.07229*, 2021.

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

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

[60] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "Mp2ml: a mixed-protocol machine learning framework for private inference," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[61] M. Izabachène, R. Sirdey, and M. Zuber, "Practical fully homomorphic encryption for fully masked neural networks," in *International Conference on Cryptology and Network Security*. Springer, 2019, pp. 24–36.

[62] Q. Lou and L. Jiang, "She: A fast and accurate deep neural network for encrypted data," *Advances in Neural Information Processing Systems*, vol. 32, pp. 10 035–10 043, 2019.

[63] A. Sanyal, M. Kusner, A. Gascon, and V. Kanade, "TAPAS: Tricks to accelerate (encrypted) prediction as a service," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4490–4499.

[64] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural netwfork on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[65] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe."

[66] Q. Lou and L. Jiang, "She: A fast and accurate deep neural network for encrypted data," https://github.com/safednn/SHE, 2019.

[67] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[68] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[69] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," https://github.com/mminelli/dinn, 2017.

[70] "Zama concrete v0.1.0," https://github.com/zama-ai/concrete, 2021, zama AI, France.

[71] X. Sui, Q. Wu, J. Liu, Q. Chen, and G. Gu, "A review of optical neural networks," *IEEE Access*, vol. 8, pp. 70 773–70 783, 2020.