nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data

Fabian Boemer[1], Yixing Lao[1], Rosario Cammarota[1], and Casimir Wierzynski[1]

Intel AI Research
San Diego, CA
`fabian.boemer@intel.com`

Abstract. Homomorphic encryption (HE)—the ability to perform computation on encrypted data—is an attractive remedy to increasing concerns about data privacy in deep learning (DL). However, building DL models that operate on ciphertext is currently labor-intensive and requires simultaneous expertise in DL, cryptography, and software engineering. DL frameworks and recent advances in graph compilers have greatly accelerated the training and deployment of DL models to various computing platforms. We introduce nGraph-HE, an extension of nGraph, Intel's DL graph compiler, which enables deployment of trained models with popular frameworks such as TensorFlow while simply treating HE as another hardware target. Our graph-compiler approach enables HE-aware optimizations– implemented at compile-time, such as constant folding and HE-SIMD packing, and at run-time, such as special value plaintext bypass. Furthermore, nGraph-HE integrates with DL frameworks such as TensorFlow, enabling data scientists to benchmark DL models with minimal overhead. [1]

Keywords: Homomorphic encryption, intermediate representation, deep learning

1 Introduction

One of the key challenges in deploying machine learning (ML) at scale is how to help data owners learn from their data while protecting their privacy. This issue has become more pressing with the advent of regulations such as the General Data Protection Regulation [54]. It might seem as though "privacy-preserving machine learning" would be a self-contradiction: ML wants data, while privacy hides data [55]. One promising solution to this problem is known as homomorphic encryption (HE). Using HE, one can perform computation on encrypted data without decrypting it. Data owners can encrypt their data with the public key, send it to a data processor that has no access to the secret key, and receive the answer to their query in encrypted form, which only the data owner can unlock with the secret key.

[1] To appear in ACM International Conference on Computing Frontiers 2019.

The idea of HE dates back to 1978 [45], and theoretical breakthroughs occurred in 2009 [27] to make the idea real but highly impractical. Further algorithmic breakthroughs have occurred since then, in tandem with the development of post-quantum cryptosystems and their implementations [7, 42] to yield HE schemes that map naturally onto vector addition and multiplication—the core of DL workloads. Recent work has shown the feasibility of evaluating convolutional neural networks using lattice-based HE cryptosystems [21, 28, 31, 34, 39, 48].

One of the biggest accelerators in DL has been the development and rapid adoption of software frameworks, such as TensorFlow [2], MXNet [16] and PyTorch [43], making use of open-source graph compilers such as Intel nGraph [50], XLA [1] and TVM [17], that allow data scientists to describe DL networks and operations at a high level while hiding details of their software and hardware implementation. By contrast, a key challenge for building privacy-preserving DL systems using HE has been the lack of such a framework. As a result, developing and deploying DL models that operate on ciphertext is currently labor intensive and forces data scientists to become experts in DL, cryptography, and software engineering.

In this work, we leverage recent work in graph compilers to overcome this challenge. Specifically, we present nGraph-HE, an HE backend to the Intel nGraph DL graph compiler that allows data scientists to train networks on the hardware of their choice in plaintext, then easily deploy these models to HE cryptosystems that operate on encrypted data. The core idea is to create a privacy-preserving hardware abstraction layer, with its own instruction set architecture (ISA) (Section 3.3) and optimization support (Section 3.4). This hides the complexity of HE from data scientists while exploiting the considerable compiler tooling and DL frameworks that the DL community has built (Figure 1). Using this approach, for example, modifying an existing TensorFlow model to operate on encrypted data becomes as easy as adding a single line of code (Appendix A.2). Indeed, the open-source release of this framework[2] has already gathered significant attention in the DL community [41, 56].

Using HE to implement DL computations imposes a number of constraints due to the mathematical requirements of HE and DL themselves, such as limited arithmetic depth and polynomial activation functions (Section 2.2). Overcoming these constraints is an area of active algorithmic research [21,28,31,34,39,48]. The contributions of this paper are along a different vector, namely, how to provide a software framework for developing privacy-preserving DL models that cleanly separates DL and HE functions (Figure 1). This will enable the DL and HE communities to improve their own technologies as independently as possible while still enjoying the advances of the other with minimal changes to the high-level code.

In this paper, we present the following:

1. We describe an efficient software framework for combining DL and HE. To our knowledge, we present the first use of a DL graph compiler and intermediate

[2] The nGraph-HE library is available under the Apache 2.0 license at https://ngra.ph/he

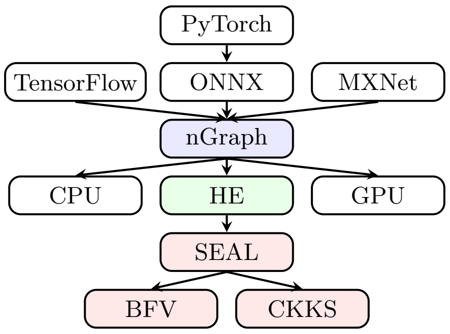


Fig. 1: Overview of the nGraph-HE software stack. nGraph-HE currently supports the SEAL encryption library [36], and the underlying cryptosystems BFV [5] and CKKS [18], and it can be extended to support additional cryptosystems.

representation (IR) to accelerate the development and deployment of privacy-preserving machine learning models.

2. We develop HE-aware graph-compiler optimizations, both at compile-time and at run-time. The compile-time optimizations include graph-level optimizations such as batch-norm folding and parallel operations through HE-SIMD packing and OpenMP parallelization. Runtime optimizations include special plaintext value bypass and ciphertext-plaintext operations.
3. We demonstrate the framework on: (1) subgraphs of DL models: general matrix-matrix multiplication (GEMM) operations, and a convolution-batch-norm operation; and (2) two convolutional neural network benchmark problems (MNIST and CIFAR-10) with different choices of encryption parameters, using Python and TensorFlow. Furthermore, we verify that the runtime overhead imposed by the additional software layers is small (0.1% of total runtime) compared to implementing these operations in C++ using HE libraries directly.

2 Background

2.1 Homomorphic encryption

What does it mean for a cryptosystem to be *homomorphic*? Informally, an encryption function E and its decryption function D are homomorphic with respect to a class of functions $\mathcal{F}$ if for any function $f \in \mathcal{F}$, we can construct a function g such that $f(x) = D\left(g(E(x))\right)$ for some set of x that we care about[3]. That is, for certain cryptosystems and target functions, it is possible to map a desired computation (the function f) on plaintext into a specific computation on

[3] We are omitting the public and secret keys that would also be arguments for the encryption and decryption functions.

ciphertext (the function g) whose result, when decrypted, matches the desired plaintext result. For a detailed review of HE, we refer the reader to [3].

Figure 2 shows how this property enables a user, Alice, to perform inference on private data using a remote, untrusted computer. The remote machine receives a ciphertext from Alice (with no decryption key), executes a function g on the ciphertext, then returns the result to Alice. Alice then decrypts the result to reveal the plaintext for $f(x)$. At no point does the remote machine gain access to Alice's unencrypted data. An analogous setup provides secure inference in the case where the function g is kept private, while the data remains unencrypted, as might occur when, for example, g corresponds to a proprietary 3rd party DL model.

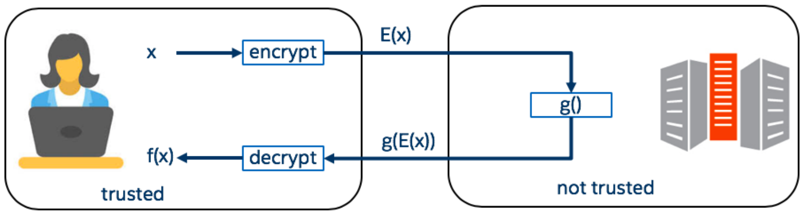


Fig. 2: Simple model of secure inference via HE.

One important property of RLWE-based HE schemes is *semantic security*, which is the inability of a computationally bounded adversary to distinguish between the ciphertexts of known plaintexts. Notably, $E(x) \neq E(y)$, even when $x = y$. This is due to random noise which is introduced during the encryption process. Without this property, a malicious remote server in Figure 2 might be able to deduce $f(x)$ in cases where g maps to a finite number of outputs, as in binary classification problem, by performing inference on inputs whose classification is already known.

2.2 Challenges of homomorphically encrypted deep learning

HE schemes are often subject to several mathematical limitations:

Supported functions. Some HE schemes only support a single algebraic operation, such as addition or multiplication. These are known as "partially homomorphic" schemes (PHE). Others schemes, called "fully homomorphic" (FHE), support two, such as addition and multiplication. Note that composing addition and multiplication suffices to construct polynomial functions, and hence polynomial approximations to non-polynomial functions such as sigmoid or ReLU[4]. Notably, this limitation prevents the exact computation of any comparison-based

[4] Going further, by building gates out of addition and multiplication over GF(2), one can in theory implement any boolean circuit, and hence any computable function.

operations, such as Max, Min, and ReLU, as well as common functions such as exponential or sigmoid. One workaround to this limitation in the case of a final softmax layer is to leave the softmax calculation to our user Alice after she decrypts the model outputs. Finally, “leveled homomorphic” schemes (LHE) support addition and multiplication, but only up to a fixed computational depth.

Computational depth. HE schemes derived from Gentry’s original lattice-based system [27] rely on noise to hide plaintext. This encryption noise tends to accumulate with each homomorphic operation, and decryption becomes impossible if this noise exceeds a threshold. One common solution to this problem is to constrain the depth of the computation and set encryption parameters accordingly. Other solutions involve noise management techniques such as *bootstrapping*, which, depending on the HE scheme, may incur significant computational costs, but can extend the computational depth indefinitely. LHE schemes do not perform bootstrapping, relying instead on the fixed computational depth of DL models.

Number fields. Most HE schemes operate over integers [29,36], while others use booleans [20] or real numbers [19]. One particular challenge in the case of integer-based schemes is scaling the magnitude of numbers by factors less than 1. Most DL models require real, i.e., non-integer, numbers, so adapting integer HE schemes typically involves mapping large integers to a fixed-point representation using a scaling factor. Preventing the scaling factor from accumulating, however, requires division by the scaling factor after each multiplication, which is not possible in all HE schemes.

Computational and memory load. The cryptographic computations required to implement HE typically consume several orders of magnitude more CPU time and memory compared to their plaintext counterparts. These costs have long been the critique of HE. A detailed response to these critiques is out of the scope of this paper, but we note that there have been dramatic improvements in this area—for example, the runtime for homomorphic inference on the seminal CryptoNets MNIST network has been reduced from 297.5s [28] to 0.03s [34] in two years (although the latter uses a hybrid scheme; see Section 5).

From a software engineering perspective, there is additional complexity: there are multiple libraries for HE [19, 20, 29, 36, 46], based on multiple HE schemes [49], and with a variety of APIs (with some notable attempts to provide uniformity [8, 46]). This diversity makes it difficult for developers to evaluate the tradeoffs of different schemes in the context of their specific applications. Moreover, the complexity of implementing DL models has led to the development of multiple DL libraries [2, 16, 33, 43, 53, 57]. Finally, and not surprisingly, no currently-available DL libraries were designed with HE in mind, and vice-versa. As a result, developers of privacy-preserving DL models have been forced either to import DL functions into HE code, or HE functions into DL code, with large code changes required if either of these library choices should change.

Given the computational and memory overhead of HE, we target inference, rather than training. The inference use case is also particularly relevant from a privacy point of view, given that statistical techniques such as differential privacy do not easily apply to the case of protecting the privacy of the query.

2.3 Mathematical Background

We provide a brief introduction to the mathematical objects used in HE. Many HE schemes are based on the assumed hardness of the Ring Learning with Errors (RLWE) problem [40], which uses polynomials whose coefficients are members of a finite field. [9]. In particular, let $N = 2^n$ for some positive integer n. N is known as the *polynomial modulus degree*. Let $\mathcal{R} = Z[x]/(x^N + 1)$ be the ring of integer-coefficient polynomials with degree at most N. We denote by $\mathcal{R}_a = \mathbb{Z}_a[x]/(x^N + 1)$ the same ring as $\mathcal{R}$, but whose coefficients are integers modulo a. The plaintexts and ciphertexts in the HE schemes we consider consist of pairs or lists of polynomials in $\mathcal{R}_t$ and $\mathcal{R}_q$, where $t, q > 0$ are known as the *plaintext modulus* and *ciphertext modulus*, respectively. In practice, t, q are often large enough that, for performance reasons, they are decomposed as $q = \prod_i q_i$, $t = \prod_i t_i$, where q_i and t_i are known as *ciphertext coefficient moduli* and *plaintext coefficient moduli*, respectively. Typically, multiplying two ciphertexts $c_1, c_2 \in \mathcal{R}_q^{j+1}$ of size $j + 1$ results in a ciphertext of larger size, up to $j^2/2$ in the BV scheme [10], and $2j + 1$ in the BFV and CKKS encryption schemes [36]. Subsequent operations on these larger ciphertexts become slower; however, an operation called *relinearization* reduces the length of each ciphertext to mitigate this slowdown. Performing relinearization requires a public *relinearization key*.

These RLWE parameters, including the polynomial modulus degree, ciphertext moduli, and plaintext moduli, are chosen to ensure a *security level*, measured in bits, where λ-bit security indicates $\sim 2^\lambda$ operations are required to break the encryption [38, 44]. The choice of λ depends on the security needs of the application, with typical values for λ being 128 bits, 192 bits, and 256 bits. The runtime requirements to mitigate several attacks on different security levels are detailed in [15]. Additionally, the parameters need to be chosen sufficiently large such that the amplification of the random noise during arithmetic operations does not render the original message unrecoverable. Specifically, each ciphertext (or plaintext) encoded in $\mathcal{R}_q$ (or $\mathcal{R}_t$) is associated with a level l with, $0 \le l \le L$, where L is maximum multiplicative depth. The multiplicative depth L is one of the parameters to the HE scheme in addition to the RLWE parameters. The HE scheme allows at most l multiplications on the ciphertext. Multiplication is typically much more expensive than addition, so the multiplicative depth of the desired computation, L_f, is an important consideration when computing on HE. Hence, for an assigned computation of a certain multiplicative depth $L \ge L_f$, HE parameters are selected to guarantee, for example, 128-bit security. The choice of parameters presents a trade-off between security to preserve data privacy, and speed of computation.

2.4 Related Work

While there has been much previous work detailing algorithmic improvements to HE for DL [21, 28, 31, 34, 39, 48], there have been only a few notable efforts to provide privacy-preserving machine learning software frameworks. PySyft [47] is a generic framework for distributed, privacy-preserving DL, built on PyTorch, that uses multi-party computation (MPC) for private computation. TF-encrypted [24] also enables private machine learning via MPC, and is built on TensorFlow. Both of these systems are tied to a specific DL framework and use MPC, not HE, which assumes a different security model. By contrast, by operating on computational graphs, nGraph-HE enables users of multiple DL frameworks (Figure 1), and requires much smaller code changes to non-HE code to invoke—potentially only one line (Appendix A.2).

There have also been some recent compiler projects around HE. The RAMPARTS project [4] translates models from the Julia language to HE operations implemented by PALISADE HE library [46]. No source code for the compiler is available, and source language support is limited to Julia. Moreover, the PALISADE library does not currently support CKKS, the HE scheme of choice for DL. The Cingulata [13] compiler uses C++ as a source language and targets a custom implementation of the Fan-Vercauteren HE scheme. Because it first translates computations into boolean circuits rather than arithmetic compute graphs, it loses performance on DL operations such as GEMM. SHEEP [32] describes an abstract ISA for HE, and includes several HE schemes and implementations. However, the low-level language and lack of compiler make it difficult to use as a data science tool.

The CHET project [25] also describes an ISA and a compiler for HE, but adopts a different approach from nGraph-HE. Whereas CHET performs compiler optimizations for HE at code generation time, just as any traditional compilation approach, nGraph-HE elevates optimizations for HE to the DL framework level, similarly to what frameworks such as TVM [17] do. Furthermore, to date, nGraph-HE is the only existing open-source framework for privacy-preserving DL. nGraph-HE's ability to support existing DL frameworks such as TensorFlow with minimal code changes is vital for data scientists, who must be able to rapidly prototype the accuracy and performance of DL models.

2.5 The power of graph compilers

DL frameworks, such as TensorFlow, MXNet and PyTorch, have greatly accelerated the development of DL models, allowing data scientists to express DL operations in high-level languages that can be easily ported from one platform to another (from a laptop to a cloud-based server, *e.g.*). Graph compilers, such as the open-source Intel nGraph, have recently been developed to attack the challenge of optimizing performance on multiple DL frameworks and hardware targets. Compiling high-level framework code into an IR—a computation graph—removes the need to generate optimized code for each (framework, hardware target) pair. Instead, in order to use a new hardware target for all frameworks, one only needs to develop optimized code for each operation in the computation graph for the targeted hardware. In addition, the graph can be an advantageous representation for reasoning about higher-level optimizations, such as fusing operations, vectorization, etc.

These advantages apply directly to expressing DL computations using HE. By treating HE schemes and the operations they support as instructions on a virtual machine [29], we can enable HE computations to a large set of DL frameworks while providing a clean separation between DL and HE technologies. Moreover, as in the case of deep learning, the graph representation facilitates various HE-specific optimizations, such as reducing computation depth and identifying opportunities for parallelism.

3 nGraph-HE

We first describe the API adopted by nGraph-HE, as well as the mapping onto two currently supported cryptosystems: BFV [5] and CKKS [18], both implemented by the SEAL encryption library [36]. We then discuss compile-time and runtime optimizations used in nGraph-HE. These include HE-specific optimizations that exploit the capabilities of the underlying cryptosystems, as well as parallelization methods to reduce execution time. Lastly, we discuss how to support additional cryptosystems.

One difficulty in providing a unified framework for HE has been the variety of APIs and supported operations for various HE schemes. Following [11], our API has three components: (1) a *cryptographic context*, which contains the static parameters of the encryption scheme, (2) a *payload representation*, which contains the data, and (3) an *assembly language*, which describes the functions implemented by the encryption scheme.

3.1 Cryptographic context

The cryptographic context stores the parameters of the cryptosystem, which consist of:

- *polynomial modulus degree* (N);
- *plaintext moduli* ($t = \prod_i t_i$);
- *ciphertext moduli* ($q = \prod_{i=1}^{L} q_i$);
- *security level* (λ);
- *HE scheme* as a unique string representation.

Depending on the cryptosystem, one or more of these parameters may not be required. The HE scheme implementations we currently support do not include bootstrapping; as such, we expect the cryptographic context to include enough ciphertext moduli to support the multiplicative depth of the DL model, i.e., $L \geq L_f$. The user will specify the cryptographic context as a command-line variable. In nGraph-HE, the "HEBackend" class stores the cryptographic context, as well as instantiations of (public, secret, relinearization) key tuples.

3.2 Payload representation

The payload representation stores the data and consists of plaintext and ciphertext representations. In nGraph-HE, the payload is stored in the "HETensor" class,

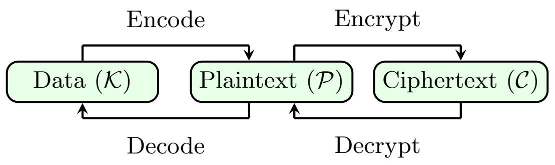


Fig. 3: Relation between payload terms.

which stores a pointer to an "HEBackend", necessary to obtain the keys. Figure 3 shows the relation between the terms in the payload representation. Specifically, we have :

- *data*: ($\mathcal{K}$). Usually $\mathcal{K} = \mathbb{R}^s$ or $\mathcal{K} = \mathbb{Z}^s$, $s \in \mathbb{N}$, a vector of numbers.
- *encode*: ($\mathcal{K} \to \mathcal{P}$). Uses the cryptographic context.
- *encrypt*: ($\mathcal{P} \to \mathcal{C}$). Uses the public key.
- *decrypt*: ($\mathcal{C} \to \mathcal{P}$). Uses the secret key.
- *decode*: ($\mathcal{P} \to \mathcal{K}$). Uses the cryptographic context.

This overall abstraction strictly generalizes the standard (encrypt, decrypt) model, since the encode and decode functions can be identity mappings. This allows us to store pre-computed plaintext values for optimization (Section 3.4).

3.3 Assembly language

We describe the assembly language of nGraph-HE in terms of nGraph operations [50]. There are four low-level operations, which are typically supported by the APIs of HE cryptosystems:

- `Add`: $(\mathcal{C} \cup \mathcal{P}) \times \mathcal{C} \to \mathcal{C}$.
- `Subtract`: $(\mathcal{C} \cup \mathcal{P}) \times \mathcal{C} \to \mathcal{C}$.
- `Multiply`: $(\mathcal{C} \cup \mathcal{P}) \times \mathcal{C} \to \mathcal{C}$. For efficiency, the implementation should use relinearization if possible.
- `Negate`: $\mathcal{C} \to \mathcal{C}$.

Additionally, HE schemes will often implement a plaintext version ($\mathcal{P} \to \mathcal{P}$ or $\mathcal{P} \times \mathcal{P} \to \mathcal{P}$) of each operation. Based on these low-level operations, nGraph-HE provides efficient parallelized implementations for the following compound operations: `AvgPool`, `Convolution`, and `Dot`. Developers can overwrite these default implementations with cryptosystem-specific optimizations.

nGraph-HE also provides implementations for the following tensor manipulation operations: `Broadcast`, `Concat`, `Pad`, `Reshape`, `Reverse`, and `Slice`. See Table 1 for a full list of supported operations and their mapping to TensorFlow operations.

Concretely, the nGraph-HE API consists of the following major components, shown in Figure 4:

Table 1: Supported operations and mapping to TensorFlow operations.

nGraph/nGraph-HE op	**TensorFlow op**
`Add`	`tf.add`
`AvgPool`	`tf.nn.avg_pool`
`Broadcast`	`tf.broadcast_to`
`Concat`	`tf.concat`
`Constant`	`tf.constant`
`Convolution`	`tf.nn.convolution`
`Dot`	`tf.matmul`
`Multiply`	`tf.multiply, tf.square`
`Negate`	`tf.negative`
`Pad`	`tf.pad`
`Reshape`	`tf.reshape`
`Reverse`	`tf.reverse`
`Slice`	`tf.slice`
`Subtract`	`tf.subtract`
`Sum`	`tf.reduce_sum`
`Parameter`	`tf.placeholder`

- *Backend*. This stores the cryptographic context, and performs graph-level optimizations. The *HESealBackend* and *HEBFVBackend* classes inherit from *HEBackend* class, which, in turn inherits from nGraph's *Backend* class.
- *Tensor*. This stores the data. *HEPlainTensor* and *HECipherTensor* inherit from *HETensor*, which in turn inherits from nGraph's *Tensor* class. HEPlainTensors store *HEPlaintext*'s, which is an abstract class from which *seal::Plaintext* inherits; HECipherTensors operate analogously.
- *Kernel*. The kernel consists of stand-alone implementations of nGraph *op*s. Each implementation operates on *HEPlaintext* and *HECiphertext* inputs, which are dynamically cast to the appropriate cryptosystem-specific type at runtime. The *Tensor* class hierarchy enables nGraph-HE to provide default implementations for each operation when no cryptosystem-specific implementation is present. This further decreases the overhead in adding a new cryptosystem to nGraph-HE.

3.4 Optimizations

One of the benefits of using a compiler approach to homomorphic computation is the ability to perform optimizations that exploit the structure of the computation, the underlying cryptosystem, and the hardware. To illustrate this benefit, we implemented three classes of optimizations of which the first is run-time, and the second two are compile-time optimizations: (1) detection of special plaintext values; (2) mapping ISA-level parallelism in the privacy-preserving abstraction layer onto the parallel structures found in HE and modern microprocessors; and (3) graph-level optimizations.

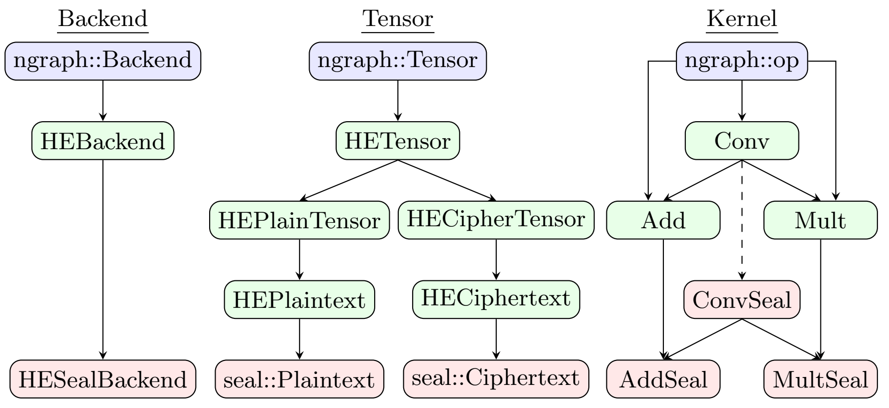


Fig. 4: Visualization of nGraph-HE architecture. Objects with the same color interact. The dotted line from Conv to ConvSeal indicates optional overriding of the default Conv implementation.

Special plaintext value bypass Operations between a ciphertext and a plaintext may arise when either the model or the data are encrypted, but not both. When performing such operations, nGraph-HE detects special values in the plaintext and, when possible, bypasses the corresponding HE operations. These runtime optimizations are HE-specific strength-reduction optimizations. Specifically, where $c \in \mathcal{C}$ is a ciphertext, and $p(i) \in \mathcal{P}$ is the plaintext encoding of i we implement:

- $c \pm p(0)$: bypass HE operations and return c;
- $c \times p(0)$: bypass HE operations and return a freshly-encrypted zero ciphertext, thereby resetting the noise budget;
- $c \times p(1)$: bypass HE operations and return c;
- $c \times p(-1)$: return the negation of c, avoiding an expensive multiply operation.

Bypassing HE operations not only reduces or resets encryption noise accumulation but also reduces runtime. One benefit of using a graph compiler is that higher-level compound operations, such as `Dot` and `Convolution`, automatically inherit the benefits of these optimizations. For instance, in a binarized neural network with binary convolution kernels, applying a `Convolution` operation will not invoke any calls to `Multiply`. To accommodate different binarization settings, we allow the user to independently enable or disable the Optimized Multiply and Optimized Addition plaintext value bypass. We demonstrate some of these runtime benefits quantitatively in Section 4.1 and Section 4.3.

Parallel operations: (1) HE-SIMD packing. Some HE schemes (including BFV and CKKS) support Single Instruction Multiple Data (SIMD) operations [51], which we refer to as “HE-SIMD” operations to avoid confusion with the usage of the "SIMD" term in computer architecture. In simple terms, a vector of payload values can be encoded and encrypted as a single ciphertext, and operations on this ciphertext are equivalent to the same HE-SIMD operation on the values in the vector individually. nGraph-HE utilizes HE-SIMD packing across the mini-batch dimension, as in [28]. Concretely, given a 4D tensor with shape (N, C, H, W) format (batch size, channels, height, width), which typically requires $N \times C \times H \times W$ ciphertexts, nGraph-HE uses HE-SIMD packing to store the tensor as a 3D tensor of $C \times H \times W$ ciphertexts, with each ciphertext packing N values. As shown in Section 4.3, running models with different mini-batch sizes (within the maximum allowed size) gives near-identical runtimes, significantly increasing the inference throughput. Analogously, loop unrolling in standard compiler optimization selects the amount of unrolling to maximize the number of utilized slots in vectorized operations. Here, increasing the batch size maximizes the number of utilized slots in each ciphertext.

Recent work [25, 34] has experimented with using HE-SIMD packing to store NCHW-formatted tensors as different 2D or 3D tensors, improving on both memory usage and runtime on small batch sizes. However, although efficient operations for convolution and dot product exist in these packing schemes, reshape and broadcast operations become more complicated, and the ciphertext slots are more difficult to utilize entirely. Hence, our choice of HE-SIMD packing represents a performance tradeoff, optimizing for throughput and simplicity of implementation over latency.

(2) OpenMP parallelization. nGraph-HE makes extensive use of OpenMP [23], an API for shared-memory programming, for parallelization. It is used in data encryption and decryption, unary and binary element-wise operations, GEMM operations, and convolution. Different from HE-SIMD packing, OpenMP parallelization is applied to non-mini-batch dimensions. For instance, to encrypt a batch of 1024 images with shape 28×28, nGraph-HE encrypts the values at the first pixel location across all 1024 images as one ciphertext with HE-SIMD packing, and does so for all 784 pixel locations in parallel with OpenMP, resulting in a total of 784 ciphertexts. OpenMP parallelization significantly reduces the inference latency of our system.

Graph-level optimizations One advantage of graph compilers is the ability to offer higher-level optimizations based on the computation graph. We briefly describe several graph optimizations analogous to standard compiler optimization of constant propagation and which are particularly relevant for HE.

– *AvgPool folding.* An AvgPool layer with window size $s_1 \times s_2$, followed by a Convolutional (Conv) layer with weights W is replaced by the equivalent ScaledMeanPool operation followed by a Conv layer with weights $W/(s_1 \times s_2)$. This reduces the multiplicative depth L_f from two to one.
– *Activation folding.* A Conv or Fully Connected (FC) layer with weights W followed by a polynomial activation of the form $ax^2 + bx + c$ is equivalent to the same Conv or FC layer with weights aW, followed by a polynomial activation of the form $x^2 + (b/a)x + (c/a)$. This reduces L_f from two to one.

– *Batch-Norm folding.* A Conv or FC layer followed by a Batch-Norm (BN) has the form:

$$z = W * x; \qquad \hat{z} = \frac{z - \mu_z}{\sqrt{\sigma_z^2 + \epsilon}}; \qquad z_{BN} = \gamma \hat{z} + \beta$$

where $\gamma, \beta, \mu_z, \sigma_z$ are all fixed during inference. A naïve implementation would require a multiplicative depth $L_f = 2$: one to compute z, and one to compute $z_{BN} = \hat{\gamma} z + \hat{\beta}$, where $\hat{\gamma} = \left(\frac{\gamma}{\sqrt{\sigma_z^2+\epsilon}} \right)$ and $\hat{\beta} = \left(\beta - \frac{\gamma \mu_Z}{\sqrt{\sigma_Z^2+\epsilon}} \right)$ are constants at inference. However, we can equivalently compute $z_{BN} = (W\hat{\gamma}) * x + \hat{\beta}$ where $(W\hat{\gamma})$ is also constant at inference. This simplified representation has multiplicative depth $L_f = 1$.

These optimizations are also possible in non-HE settings; for instance, BN folding is implemented in TensorFlow. However, the reduction in L_f makes these optimizations especially useful in HE models. For instance, AvgPool folding is used in the CryptoNets model [28]. See Section 4.3 and Section 4.2 for examples of BN folding.

3.5 Ciphertext-plaintext operations

HE implementations of ciphertext-plaintext operations $\mathcal{C} \times \mathcal{P} \to \mathcal{C}$ are typically much faster than implementations of ciphertext-ciphertext operations $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$. To take advantage of this performance gain, nGraph-HE allows for three distinct computing paradigms, based on the privacy needs of the application.

– *Encrypted data, unencrypted model.* This use case occurs when private data are obtained from medical patients, while the model is kept on a remote server. This paradigm is the fastest, as it allows the most number of $\mathcal{C} \times \mathcal{P}$ operations.
– *Encrypted model, unencrypted data.* This is the case when a company deploys a proprietary model to untrusted servers.
– *Encrypted data, encrypted model.* Here, both the model and data are kept encrypted for most privacy, at the cost of the slowest runtime. This use case might occur when a company deploys a proprietary model to perform computations on sensitive data on untrusted hardware.

For debugging purposes, nGraph-HE also offers each operation in plaintext: $\mathcal{P} \times \mathcal{P} \to \mathcal{P}$ or $\mathcal{P} \to \mathcal{P}$.

3.6 Adding a new cryptosystem

Currently, nGraph-HE supports two cryptosystems, each implemented by the SEAL encryption library: BFV and CKKS. To support another cryptosystem, one simply needs to implement the storage model and the low-level operations in the assembly language instructions described above. Most HE cryptosystems already include similar APIs, so the implementation is usually straightforward. As shown in Section 3.3, nGraph-HE provides default implementations for higher-level compound ops such as `Dot` and `Convolution`, which may be overridden with more efficient cryptosystem-specific implementations by the developer.

3.7 DL Framework Integration

A critical aspect of a new software library is the ease of adoption. nGraph-HE leverages Intel nGraph-TensorFlow [14] for seamless integration with the TensorFlow DL library [2]. Modifying existing TensorFlow code to use nGraph-HE requires adding only a single line of code. See Appendix A.2 for a full example. This makes nGraph-HE extremely easy to use. To use models from other DL frameworks, such as MXNet, ONNX, and PyTorch, nGraph-HE users must first export the DL model to nGraph's serialized format.

nGraph-HE supports most operations commonly found in neural networks. Table 1 shows the full list of operations currently supported by nGraph-HE, and the corresponding translation from TensorFlow operations. Notably absent is the support for MaxPool and ReLU operations. This is because HE supports only addition and multiplication, through which MaxPool and ReLU operations cannot be expressed.

4 Evaluation

We tested nGraph-HE on a dual-socket Intel Xeon Platinum 8180 2.5GHz system with 376GB of RAM running Ubuntu 16.04. We used SEAL's implementation of CKKS and floating-point numbers for these measurements, although we have also tested nGraph-HE with SEAL's BFV implementation. We report two main findings. First, we leverage our compiler framework to implement HE-specific optimizations on small computation graphs. Second, we demonstrate the ease of implementing convolutional neural networks using a popular DL framework (TensorFlow), on both the MNIST [37] and CIFAR-10 [35] datasets. For the MNIST dataset, we additionally:

– verify that the additional software layers through nGraph-HE to the underlying HE library impose minimal overhead;
– demonstrate the HE-SIMD packing (Section 3.4) and special plaintext value bypass (Section 3.4) optimizations;
– show the runtime dependence on three computing paradigms: encrypted data, encrypted model, encrypted data and model.

For the CIFAR-10 dataset, we also demonstrate the BN folding optimization.

4.1 GEMM operations

We first tested nGraph-HE on general matrix-matrix multiplication (GEMM) operations, since these form the backbone of DL workloads. Figure 5 shows the runtime of computing $AB + C$, where A, B, and C are $n \times n$ matrices of random integers, and where $A \in \mathcal{C}$, while $B, C \in \mathcal{P}$. (This corresponds to the encrypted data, unencrypted model use case where, A contains a user's data while B and C are model weights.) To demonstrate two different parameter settings, we set the polynomial modulus degree to $N = 2^{13}$ and $N = 2^{14}$, and used SEAL's default ciphertext modulus for $\lambda = 128$-bit security.

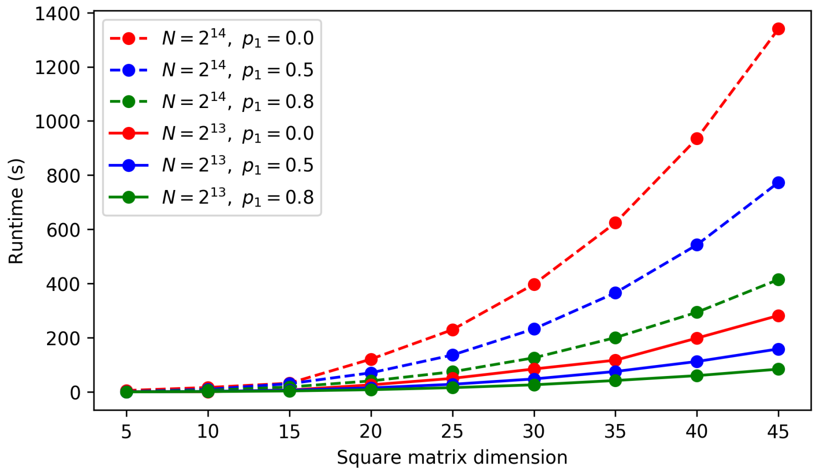


Fig. 5: Single-threaded runtime on GEMM operations as a function of matrix size, polynomial modulus, and sparsity.

To illustrate the power of enabling HE using graph compilers, we perform the Optimized Multiply special plaintext value bypass on the $\mathcal{P} \times \mathcal{C}$ multiplication operation (Section 3.4). We then measured the runtime savings by randomly setting 50% and 80% of the B matrix to 1. These results correspond to the $p_1 = 0.5, 0.8$ curves in Figure 5. Because multiplication is more expensive than in most HE schemes, the runtime improvement is significant. The larger point, however, is that providing HE in the context of a graph compiler enables developers to provide HE-specific optimizations to the backend while data scientists continue to use the DL framework of their choice, treating HE as just another (virtual) hardware target.

4.2 Graph-level optimizations

To demonstrate the utility of graph-level optimizations, we show an example of BN folding. We perform Convolution on a 3-channel 10×10 input of shape (1,3,10,10) using 4 kernels per channel, each of size 5×5, followed by BN. We consider two choices of parameters, each with security level $128 < \lambda < 192$: (1) $N = 2^{14}$, 7 50-bit coefficient moduli; (2) $N = 2^{13}$, 4 50-bit coefficient moduli.[5] Table 2 shows a moderate ∼4% decrease in runtime using BN folding, as expected. The larger point, however, is that this HE-specific graph-level optimization reduces the multiplicative depth L_f, enabling smaller encryption parameters, thereby greatly improving both runtime (∼4x) and memory usage.

Table 2: Single-threaded runtimes on Conv-BN function when encrypting the data, using nGraph-HE directly. Runtimes are averaged across 10 trials.

N	**BN folding**	**L_f**	**Runtime (s)**		
			Conv	**BN**	**Total**
2^{14}	✗	2	130.83 ± 1.14	6.28 ± 0.12	137.24 ± 1.21
2^{14}	✓	1	130.57 ± 1.57	0.25 ± 0.01	130.97 ± 1.57
2^{13}	✓	1	33.06 ± 0.68	0.06 ± 0.00	33.16 ± 0.68

4.3 Neural networks

Next, to demonstrate the ease of using nGraph-HE, we implement neural networks on the standard MNIST dataset [37], as well as the more challenging CIFAR-10 dataset [35].

MNIST The MNIST dataset consists of handwritten digits, 60,000 for training, and 10,000 for testing, and is a standard benchmark for DL on HE. The original CryptoNets network [28] is the standard HE-friendly network for MNIST, with architecture given in Appendix A.1. Appendix A.2 shows the code to implement this network, which notably differs from the native TensorFlow code by just one line. We achieve an accuracy of ∼99%, matching that reported in [28].

One concern with adding software abstractions is the runtime overhead. To measure this, we timed the network executing the TensorFlow code with nGraph-HE as the backend. This incurs the overhead of TensorFlow, the nGraph-TensorFlow bridge, and nGraph IR compilation. Within this execution, we separately time the sections that are also used in the execution of a C++ application that executes the serialized network natively.

Table 3 shows the runtimes of these experiments, using $N = 2^{13}$, $N = 2^{14}$. We use SEAL's first 7 30-bit ciphertext coefficient moduli for CKKS (i.e., $q = \prod_{i=1}^{7} q_i$, with each q_i consisting of 30-bits), for security levels $128 < \lambda < 192$ and $\lambda > 256$,

[5] A given set of encryption parameters achieves security level λ if the coefficient modulus is smaller than SEAL's default coefficient choice at the same (N, λ) pair [36]. For instance $7 \times 50 = 350$, which is between SEAL's 305-bit ($N = 2^{14}, \lambda = 128$) and 438-bit modulus ($N = 2^{14}, \lambda = 192$), hence we achieve security level $128 < \lambda < 192$.

respectively. Note that the differences in times between the fourth and fifth columns (0.02s and 0.03s), which capture the overhead of graph compilation and bridging nGraph to TensorFlow, represent less than 0.1% of overall runtime.

Table 3: Runtimes on CryptoNets network with and without the overhead of TensorFlow integration and graph compilation. Runtimes are averaged across 10 trials. Amortized runtimes are reported per image using batch size $N/2$ for maximum throughput and HE-SIMD slot utilization.

N	L_f	Acc. (%)	Runtime (s)		
			nGraph-HE	**TF+nGraph-HE**	**Amortized**
2^{13}	5	$\sim$99	16.70 ± 0.23	16.72 ± 0.23	0.004
2^{14}	5	$\sim$99	41.91 ± 1.58	41.94 ± 1.58	0.005

Another benefit of using a graph compiler with HE is that the computation graphs provide opportunities to identify parallelism that can be exploited by some HE schemes, such as the ability to perform "HE-SIMD" operations on vectors of payload values (Section 3.4). We implemented this capability and demonstrate it on the CryptoNets network. Figure 6 shows the CryptoNets inference runtime using batch sizes of 1 to 4096 for $N = 2^{13}, 2^{14}$. We picked a maximum batch size of $2^{12} = 4096$ because CKKS performs HE-SIMD operations on at most $N/2$ packed values. Note that runtimes are independent of batch size, for each step in running the network. Batching increases throughput significantly: for example, for the $N = 2^{13}$ case, using a batch size of 4096 leads to an amortized runtime of 4.1ms per image, compared to 16.7s for a batch size of 1.

Another optimization nGraph-HE implements is $\mathcal{C} \times \mathcal{P}$ operations, which are typically much faster than $\mathcal{C} \times \mathcal{C}$ operations, and enable the three computing paradigms (encrypted data, model, or both) discussed in Section 3.5. Using the CryptoNets network and same cryptographic context, Table 4 shows the fastest runtime is achieved when just the data is encrypted. Encrypting the model incurs $\sim$3.2x runtime penalty, whereas encrypting the data and the model incurs $\sim$3.6x runtime penalty. Users can switch between the computing paradigms, enabling users to measure the privacy-performance tradeoff.

Finally, to demonstrate the benefit of special plaintext value bypass (Section 3.4), we implement a binarized neural network. We adapt the CryptoNets network by binarizing each weight in the FC and Conv layers to $\{-1, 1\}$, using the approach in [22]. To mitigate the diminished accuracy, we further add a non-binarized BN layer after each FC and Conv layer. BN-folding is disabled to preserve the binarization of the FC and Conv weights.

We consider two choices of parameters: 1) $N = 2^{14}$, 9 30-bit coefficient moduli, with security $192 < \lambda < 256$; 2) $N = 2^{13}$, 7 30-bit coefficient moduli, with security $128 < \lambda < 192$. As shown in Table 5, enabling the Optimized Multiply special

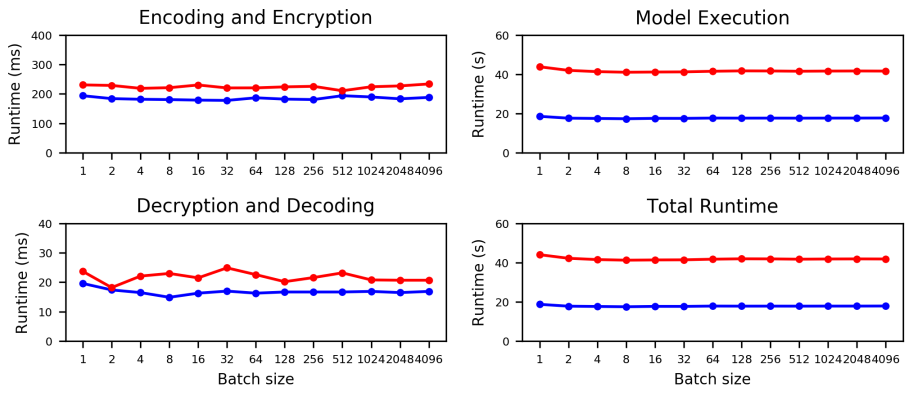


Fig. 6: Runtimes on pre-compiled CryptoNets network with HE-SIMD packing for different batch sizes, for $N = 2^{13}$ (—) and $N = 2^{14}$ (—).

Table 4: Runtimes on CryptoNets network when encrypting the data, the model, or both, using the TensorFlow nGraph-HE integration. Runtimes are in seconds and averaged across 10 trials.

N	Encrypt		
	Data	**Model**	**Data and model**
2^{13}	16.7 ± 0.2	53.3 ± 1.0	59.5 ± 1.7
2^{14}	41.9 ± 1.6	128.2 ± 1.2	142.6 ± 9.9

plaintext value bypass provides a moderate ∼1.2x speedup. However, a more significant ∼3.7x runtime speedup arises due to the lower multiplicative depth L_f, enabling a smaller choice of N. The runtime of 14.8s is faster than 16.7s in the original CryptoNets network, at the cost of reduced accuracy.

Table 5: Runtimes on binarized CryptoNets network when encrypting the data, using the TensorFlow nGraph-HE integration. Runtimes are averaged across 10 trials.

N	L	L_f	**Optimized Mult.**	**Runtime (s)**	**Acc. (%)**
2^{14}	9	8	✗	55.2 ± 2.3	96.9 ± 0.5
2^{14}	9	5	✓	45.2 ± 1.3	96.9 ± 0.5
2^{13}	7	5	✓	14.8 ± 0.9	96.9 ± 0.5

CIFAR-10 The CIFAR-10 dataset is a standard image classification dataset consisting of 60,000 color images of shape $32 \times 32 \times 3$, of which 50,000 are used for training, and 10,000 are used for testing, and with 6,000 examples for each of 10 different classes. The larger image size and color channels make CIFAR-10 a significantly more challenging task than MNIST. There is currently no seminal CIFAR-10 HE-friendly network as there is for MNIST. Due to the use of unbounded polynomial activations, numerical overflow during training is prevalent, although gradient clipping and BN somewhat mitigate this effect. We implement a CIFAR-10 network with architecture given in Appendix A.1. To demonstrate the versatility of our framework, we train the CIFAR-10 network in four different configurations, by toggling two independent settings:

- *BN*. If enabled, a BN layer is added after each Conv layer. During training, we use batch size $n = 128$; during inference, we use HE-SIMD packing to enable batch size $n = 8192$.
- *Trained activations*. If enabled, each polynomial activation is of the form $ax^2 + bx$, with a initialized to 0, and b initialized to 1, and with a, b updated during training. If disabled, each polynomial activation is $0.125x^2+0.5x+0.25$, following the approach in [21].

Furthermore, to prevent numerical overflow during training, we clip the gradients to $[-0.25, 0.25]$. To demonstrate the advantage of our graph-level optimizations, we toggle the BN folding optimization where BN is used. The CIFAR-10 network has a multiplicative depth $L_f = 8$, which is significantly deeper than the CryptoNets network, with $L_f = 5$. In order to accommodate this additional depth, we choose 10 30-bit ciphertext coefficient moduli, and $N = 2^{14}$ for security level $\lambda = 192$ [36]. When BN folding is disabled, L_f increases to 10. Accordingly, we use 11 30-bit ciphertext moduli for a reduced security level of $128 < \lambda < 192$.

Table 6 shows the runtimes of the CIFAR-10 network, which are significantly higher than the MNIST CryptoNets network, due to the increased complexity of the model and dataset. BN provides a significant increase in accuracy, as polynomial activations are constrained within a narrower range. We observe L_f is constant when BN folding optimization is enabled. Enabling BN-folding reduces L_f from 10 to 8, with negligible speedup. However, the reduced multiplicative depth allows for use of fewer ciphertext moduli, which provides a $\sim$1.2x speedup.

5 Conclusion and Future Work

We have presented nGraph-HE, a backend to the Intel nGraph DL compiler, which enables DL on homomorphically encrypted data. nGraph-HE supports a variety of DL frameworks such as TensorFlow to allow easy adoption by data scientists. We have demonstrated the capability of nGraph-HE to implement networks on MNIST and CIFAR-10 with minimal computational and code overhead. Furthermore, we demonstrated several optimizations, including special plaintext value bypass, HE-SIMD packing, graph-level optimizations, and plaintext operations. The data scientist can take advantage of these optimizations with only minimal changes to their code, enabling rapid prototyping and development.

Table 6: Runtimes on CIFAR-10 network when encrypting the data, using the direct nGraph-HE integration. Runtimes and accuracies are averaged across 10 trials. Amortized runtimes are per image, using batch size $N/2$ for maximum throughput and HE-SIMD slot utilization.

L	L_f	BN/fold	Act.	Accuracy (%)	Runtime (s)	
					Total	Amortized
11	10	✓/ ✗	Train	62.1 ± 6.4	1628 ± 37	0.199
11	8	✓/ ✓	Train	62.1 ± 6.4	1637 ± 42	0.200
10	8	✓/ ✓	Train	62.1 ± 6.4	1350 ± 22	0.165
11	10	✓/ ✗	Fix	62.2 ± 3.5	1641 ± 32	0.200
11	8	✓/ ✓	Fix	62.2 ± 3.5	1651 ± 33	0.202
10	8	✓/ ✓	Fix	62.2 ± 3.5	1359 ± 19	0.166
10	8	✗	Tr	55.6 ± 6.7	1321 ± 20	0.161
10	8	✗	Fix	57.8 ± 1.3	1324 ± 13	0.161

Looking ahead, an additional benefit of using graph compilers in the context of HE is the ability to extract the computational (especially multiplicative) depth of the computation, which is needed to set the security parameters of the HE scheme. A useful extension of this work, therefore, would be to enable automatic selection of HE parameters at compile time as a function of desired security level. Another area for future work is to incorporate recent optimizations for matrix operations in HE [34]. Finally, we would like to extend this framework so that it can also include hybrid schemes that combine HE with multi-party-computation (MPC), such as garbled circuits [6, 52], or oblivious transfer [12, 26, 30]. Such hybrids have been shown [34] to deliver much faster performance at the expense of higher communication costs. The optimal decomposition of a DL workload into HE and MPC stages could be determined at compile time and would be greatly facilitated by access to the underlying computation graph.

References

1. Xla overview (2019), https://www.tensorflow.org/xla/overview
2. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: a system for large-scale machine learning. In: OSDI. vol. 16, pp. 265–283 (2016)
3. Acar, A., Aksu, H., Uluagac, A.S., Conti, M.: A survey on homomorphic encryption schemes: theory and implementation. ACM Computing Surveys (CSUR) **51**(4), 79 (2018)
4. Archer, D.: Ramparts: Rapid machine-learning processing applications and reconfigurable targeting of security (2018), https://galois.com/project/ramparts/
5. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: International Conference on Selected Areas in Cryptography. pp. 423–442. Springer (2016)
6. Bellare, M., Hoang, V.T., Keelveedhi, S., Rogaway, P.: Efficient garbling from a fixed-key blockcipher. In: 2013 IEEE Symposium on Security and Privacy. pp. 478–492. IEEE (2013)
7. Bernstein, D.J., Lange, T.: Post-quantum cryptography-dealing with the fallout of physics success. IACR Cryptology ePrint Archive **2017**, 314 (2017)
8. Boura, C., Gama, N., Georgieva, M.: Chimera: a unified framework for b/fv, tfhe and heaan fully homomorphic encryption and predictions for deep learning. IACR Cryptology ePrint Archive **2018**, 758 (2018)
9. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Annual cryptology conference. pp. 505–524. Springer (2011)
10. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. SIAM Journal on Computing **43**(2), 831–871 (2014)
11. Brenner, M., Dai, W., Halevi, S., Han, K., Jalali, A., Kim, M., Laine, K., Malozemoff, A., Paillier, P., Polyakov, Y., Rohloff, K., Savaş, E., Sunar, B.: A standard api for rlwe-based homomorphic encryption. Tech. rep., HomomorphicEncryption.org, Redmond WA, USA (July 2017)
12. Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., Schneider, T.: Hycc: Compilation of hybrid protocols for practical secure computation. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 847–861. ACM (2018)
13. Carpov, S., et al.: Cingulata (2018), https://github.com/CEA-LIST/Cingulata
14. Chakraborty, A., Proctor, A., et al.: Intel(r) ngraph(tm) compiler and runtime for tensorflow* (2018), https://github.com/NervanaSystems/ngraph-tf
15. Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Hoffstein, J., Lauter, K., Lokam, S., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Security of homomorphic encryption. Tech. rep., HomomorphicEncryption.org, Redmond WA, USA (July 2017)
16. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
17. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al.: TVM: An automated end-to-end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 578–594 (2018)

18. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. IACR Cryptology ePrint Archive **2018**, 931 (2018), https://eprint.iacr.org/2018/931
19. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 409–437. Springer (2017)
20. Chillotti, I., Gama, N., Georgieva, M., Izabachene, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 3–33. Springer (2016)
21. Chou, E., Beal, J., Levy, D., Yeung, S., Haque, A., Fei-Fei, L.: Faster cryptonets: Leveraging sparsity for real-world encrypted inference. arXiv preprint arXiv:1811.09953 (2018)
22. Courbariaux, M., Bengio, Y., David, J.P.: Binaryconnect: Training deep neural networks with binary weights during propagations. In: Advances in neural information processing systems. pp. 3123–3131 (2015)
23. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
24. Dahl, M., Mancuso, J., Dupis, Y., Decoste, B., Giraud, M., Livingstone, I., Patriquin, J., Uhma, G.: Private machine learning in tensorflow using secure computation. arXiv preprint arXiv:1810.08130 (2018)
25. Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., Mytkowicz, T.: Chet: Compiler and runtime for homomorphic evaluation of tensor programs. arXiv preprint arXiv:1810.00845 (2018)
26. Demmler, D., Schneider, T., Zohner, M.: Aby-a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
27. Gentry, C., Boneh, D.: A fully homomorphic encryption scheme, vol. 20. Stanford University, Stanford (2009)
28. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In: International Conference on Machine Learning. pp. 201–210 (2016)
29. Halevi, S., Shoup, V.: Algorithms in HElib. In: International Cryptology Conference. pp. 554–571. Springer (2014)
30. Henecka, W., Sadeghi, A.R., Schneider, T., Wehrenberg, I., et al.: Tasty: tool for automating secure two-party computations. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 451–462. ACM (2010)
31. Hesamifard, E., Takabi, H., Ghasemi, M.: Cryptodl: Deep neural networks over encrypted data. arXiv preprint arXiv:1711.05189 (2017)
32. Institute, A.T.: Sheep is a homomorphic encryption evaluation platform (2018), https://github.com/alan-turing-institute/SHEEP
33. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia. pp. 675–678. ACM (2014)
34. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: `GAZELLE`: A low latency framework for secure neural network inference. In: 27th (USENIX) Security Symposium (USENIX) Security 18). pp. 1651–1669 (2018)
35. Krizhevsky, A., Nair, V., Hinton, G.: The cifar-10 dataset. online: http://www. cs. toronto. edu/kriz/cifar. html (2014)
36. Laine, K.: Simple encrypted arithmetic library-SEAL (v3.1). Tech. rep., Technical report, June (2018)

37. LeCun, Y.: The mnist database of handwritten digits. http://yann. lecun. com/exdb/mnist/ (1998)
38. Lindner, R., Peikert, C.: Better key sizes (and attacks) for lwe-based encryption. In: Cryptographers' Track at the RSA Conference. pp. 319–339. Springer (2011)
39. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via miniONN transformations. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 619–631. ACM (2017)
40. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230 (2012), https://eprint.iacr.org/2012/230
41. Mancuso, J.: Privacy-preserving machine learning 2018: A year in review (2019), https://medium.com/dropoutlabs/privacy-preserving-machine-learning-2018-a-year-in-review-b6345a95ae0f
42. Nejatollahi, H., Dutt, N.D., Ray, S., Regazzoni, F., Banerjee, I., Cammarota, R.: Post-quantum lattice-based cryptography implementations: A survey. ACM Comput. Surv. **51**(6), 129:1–129:41 (2019)
43. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch. In: NIPS-W (2017)
44. van de Pol, J., Smart, N.P.: Estimating key sizes for high dimensional lattice-based systems. In: IMA International Conference on Cryptography and Coding. pp. 290–303. Springer (2013)
45. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of secure computation **4**(11), 169–180 (1978)
46. Rohloff, K.: The palisade lattice cryptography library (2018), https://git.njit.edu/palisade/PALISADE
47. Ryffel, T., Trask, A., Dahl, M., Wagner, B., Mancuso, J., Rueckert, D., Passerat-Palmbach, J.: A generic framework for privacy preserving deep learning. arXiv preprint arXiv:1811.04017 (2018)
48. Sanyal, A., Kusner, M., Gascon, A., Kanade, V.: Tapas: Tricks to accelerate (encrypted) prediction as a service. In: International Conference on Machine Learning. pp. 4497–4506 (2018)
49. Sathya, S.S., Vepakomma, P., Raskar, R., Ramachandra, R., Bhattacharya, S.: A review of homomorphic encryption libraries for secure computation. arXiv preprint arXiv:1812.02428 (2018)
50. Scott Cyphers, A.K.B., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., Kimball, R., Knight, J., Korovaiko, N., Kumar, V., Lao, Y., Lishka, C.R., Menon, J., Myers, J., Narayana, S.A., Procter, A., Webb, T.J.: Intel ngraph: An intermediate representation, compiler, and executor for deep learning. CoRR **abs/1801.08058** (2018), http://arxiv.org/abs/1801.08058
51. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations (2011), https://eprint.iacr.org/2011/133
52. Songhori, E.M., Hussain, S.U., Sadeghi, A.R., Schneider, T., Koushanfar, F.: Tinygarble: Highly compressed and scalable sequential garbled circuits. In: 2015 IEEE Symposium on Security and Privacy. pp. 411–428. IEEE (2015)
53. Tokui, S., Oono, K., Hido, S., Clayton, J.: Chainer: a next-generation open source framework for deep learning. In: Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS). vol. 5, pp. 1–6 (2015)

54. Voigt, P., Von dem Bussche, A.: The EU General Data Protection Regulation (GDPR), vol. 18. Springer (2017)
55. Wierzynski, C., Wen, A.: Advancing both a.i. and privacy is not a zero-sum game (2018), http://fortune.com/2018/12/27/ai-privacy
56. Wiggers, K.: A.i. weekly: 8 takeaways from neurips 2018 (2018), https://venturebeat.com/2018/12/07/ai-weekly-8-takeaways-from-neurips-2018/
57. Yang, Y., Qiao, L., et al.: Baidu parallel distributed deep learning (2018), https://github.com/PaddlePaddle/Paddle

A Appendix

A.1 Network Architectures

For each architecture, n indicates the batch size.

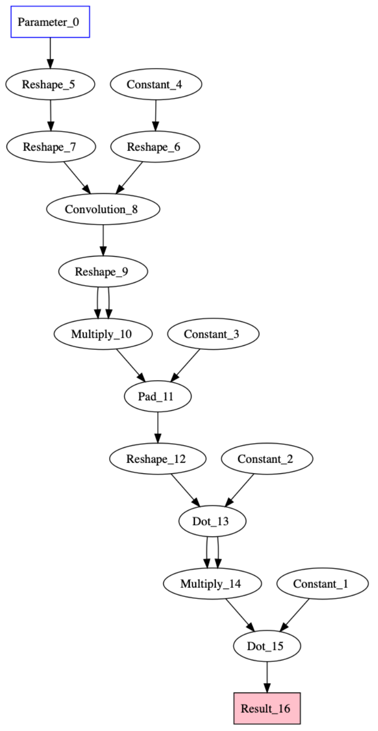
- CryptoNets, with activation $Act(x) = x^2$.
  1. *Conv.* [Input: $n \times 28 \times 28$; stride: 2; window: 5×5; filters: 5, output: $n \times 845$] + *Act.*
  2. *FC.* [Input: $n \times 845$; output: $n \times 100$] + *Act.*
  3. *FC.* [Input: $n \times 100$; output: $n \times 10$].
- Binarized CryptoNets, with activation $Act(x) = x^2$.
  1. *BinaryConv.* [Input: $n \times 28 \times 28$; stride: 2; window: 5×5; filters: 5, output: $n \times 845$] + *BN* + *Act.*
  2. *BinaryFC.* [Input: $n \times 845$; output: $n \times 100$] + *BN* + *Act.*
  3. *BinaryFC.* [Input: $n \times 100$; output: $n \times 10$] + *BN.*
- CIFAR-10 network, with polynomial activation.
  1. *Conv.* [Input: $n \times 32 \times 32 \times 3$; stride: 2; window: 5×5; filters: 40, output: $n \times 40 \times 16 \times 16$] + (BN) + *Act.*
  2. *AvgPool.* [Input: $n \times 32 \times 32 \times 3$; stride: 2; window: 5×5, filters: 40, output: $n \times 40 \times 8 \times 8$].
  3. *Conv.* [Input: $n \times 40 \times 8 \times 8$; stride: 1; window: 3×3; filters: 80, output: $n \times 80 \times 8 \times 8$] + (BN) + *Act.*
  4. *FC.* [Input: $n \times 5120$, output: $n \times 10$].

A.2 CryptoNets Inference

Figure 7 shows a full example of performing inference on the CryptoNets model. Note that the only modification required to enable nGraph-HE is the addition of the `import ngraph_bridge` line.

```
"""CryptoNets MNIST classifier"""
import ngraph_bridge  # <-- enable nGraph-HE
import numpy as np
from tensorflow.examples.tutorials.mnist
     import input_data
import tensorflow as tf

batch_size = 4096

mnist = input_data.read_data_sets(
    '/tmp/tensorflow/mnist/input_data',
         one_hot=True)

# Create inference network
parameter_0 = tf.placeholder(tf.float32, [None
     , 784])
reshape_5_7 = tf.reshape(parameter_0, [-1, 28,
      28, 1])
constant_4 = tf.constant(
    np.loadtxt('W_conv1.txt', dtype='f').
         reshape([5, 5, 1, 5]))
convolution_8 = tf.nn.conv2d(
    reshape_5_7, constant_4, strides=[1, 2, 2,
          1], padding='VALID')
multiply_10 = tf.square(convolution_8)
constant_3 = tf.constant([[0, 0], [0, 1], [0,
     1], [0, 0]])
pad_11 = tf.pad(multiply_10, constant_3)
reshape_12 = tf.reshape(pad_11, [-1, 845])
constant_2 = tf.constant(
    np.loadtxt("W_squash.txt", dtype='f').
         reshape([845, 100]))
dot_13 = tf.matmul(reshape_12, constant_2)
multiply_14 = tf.square(dot_13)
constant_1 = tf.constant(np.loadtxt('W_fc2.txt
     ', dtype='f').reshape([100, 10]))
y_conv = tf.matmul(multiply_14, constant_1)

# Run network
with tf.Session() as sess:
    x_test = mnist.test.images[:batch_size]
    y_conv_val = y_conv.eval(feed_dict={
         parameter_0: x_test})
```

(a) Python code to execute a trained CryptoNets model using TensorFlow



(b) Computational graph generated from the Python code by the nGraph compiler

Fig. 7: Source code and intermediate representation of the MNIST CryptoNets network.

B nGraph-HE Artifact Appendix

Code and runtime artifacts to replicate runtime results are publicly available at https://github.com/NervanaSystems/he-transformer/tree/v0.2-benchmarks-2

Specifically, the *benchmarks* folder contains detailed instructions on how to replicate the results, including our own runtime results from which the tables and figures were created.

Performance analysis completed on Jan 16 - Mar 21, 2019 using a Xeon Platinum 8180 platform with 112 CPUs operating at 2.5Ghz, 2 sockets, and 376GB of RAM running HE Transformer (v0.2) with nGraph-tf (v0.9.0) and nGraph (v0.11.0) on Ubuntu 16.04.4 LTS.