CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing

Roshan Dathathri
University of Texas at Austin, USA
roshan@cs.utexas.edu

Olli Saarikivi
Microsoft Research, USA
olsaarik@microsoft.com

Hao Chen
Microsoft Research, USA
haoche@microsoft.com

Kim Laine
Microsoft Research, USA
kilai@microsoft.com

Kristin Lauter
Microsoft Research, USA
klauter@microsoft.com

Saeed Maleki
Microsoft Research, USA
saemal@microsoft.com

Madanlal Musuvathi
Microsoft Research, USA
madanm@microsoft.com

Todd Mytkowicz
Microsoft Research, USA
toddm@microsoft.com

Abstract

Fully Homomorphic Encryption (FHE) refers to a set of encryption schemes that allow computations on encrypted data without requiring a secret key. Recent cryptographic advances have pushed FHE into the realm of practical applications. However, programming these applications remains a huge challenge, as it requires cryptographic domain expertise to ensure correctness, security, and performance.

CHET is a domain-specific optimizing compiler designed to make the task of programming FHE applications easier. Motivated by the need to perform neural network inference on encrypted medical and financial data, CHET supports a domain-specific language for specifying tensor circuits. It automates many of the laborious and error prone tasks of encoding such circuits homomorphically, including encryption parameter selection to guarantee security and accuracy of the computation, determining efficient tensor layouts, and performing scheme-specific optimizations.

Our evaluation on a collection of popular neural networks shows that CHET generates homomorphic circuits that outperform expert-tuned circuits and makes it easy to switch across different encryption schemes. We demonstrate its scalability by evaluating it on a version of SqueezeNet, which to the best of our knowledge, is the deepest neural network to be evaluated homomorphically.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314628

CCS Concepts • **Software and its engineering** → **Compilers**; • **Security and privacy** → *Software and application security*; • **Computer systems organization** → *Neural networks.*

Keywords Homomorphic encryption, domain-specific compiler, neural networks, privacy-preserving machine learning

ACM Reference Format:
Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314628

1 Introduction

Fully Homomorphic Encryption (FHE) provides an exciting capability of performing computation on encrypted data without requiring the decryption key. This holds the promise of enabling rich privacy-preserving applications where the clients offload their data storage and computation to a public cloud without having to trust either the cloud software vendor, the hardware vendor, or a third party with their key.

The first FHE scheme was proposed by Gentry et al. [22] in 2009. While a theoretical breakthrough, a direct implementation of this scheme was considered impractical. Cryptographic innovations in the past decade have since made significant progress in both performance and supporting richer operations. Original FHE schemes only supported Boolean operations [22]. Subsequent schemes [7, 21] supported integer operations, thereby avoiding the need to encode arithmetic operations as Boolean circuits. Recently, Cheon et al. [15, 16] proposed an FHE scheme that efficiently supports fixed-point arithmetic extending the reach of FHE to domains such as machine learning. Together with these innovations, optimized open-source implementations of FHE

schemes, such as SEAL [38] and HEAAN [27], have made FHE more accessible.

Nevertheless, building effective FHE applications today requires direct involvement of a cryptographic expert intricately familiar with the encryption schemes. Current FHE schemes work by introducing noise during encryption that is subsequently removed during decryption. The amount of noise introduced during encryption and each intermediate operation is controlled by a set of encryption parameters that are set manually. Setting these parameters low can make the encryption insecure. On the other hand, setting them large can increase the size of encrypted text and increase the cost of homomorphic operations. Moreover, when the accumulated noise exceeds a bound determined by these parameters, the encrypted result becomes corrupted and unrecoverable.

As individual homomorphic operations are orders of magnitude more expensive than equivalent plaintext operations, amortizing the cost of individual FHE operations requires utilizing the vectorization capabilities (also called as "batching" in the FHE literature) of the encryption schemes. Different ways of mapping application parallelism onto ciphertext vectors can result in different circuits each of which require a subsequent retuning of encryption parameters for correctness, security, and performance. As a result, developing FHE applications today is laboriously hard.

In many ways, programming FHE applications today is akin to low-level programming against a custom hardware with counter-intuitive tradeoffs. As such, our hypothesis is that a compiler that raises the programming abstraction while hiding and automating many of the manual tasks can make FHE programming easier and scalable. More importantly, by systematically exploring the various performance tradeoffs, a compiler can generate far more efficient code than those produced manually by experts.

This paper evaluates this hypothesis with CHET, a compiler for homomorphic tensor programs. The input domain is primarily motivated by the need to perform neural-network inference on privacy-sensitive data, such as medical images and financial data. For privacy reasons, these applications are performed on-premise today. Offloading the storage and computation to a cloud provider would not only simplify the operational cost of maintaining on-premise clusters but also dramatically reduce the data-management cost of protecting sensitive data from unauthorized accesses both from within and outside the organization. Thus, FHE enables an attractive way to move these applications to the cloud without enlarging the trust domain beyond the organization owning the data.

Given a tensor circuit, CHET compiles the circuit into an executable that can be linked with FHE libraries such as SEAL [15, 38] or HEAAN [16, 27]. The compiler uses a cost model of homomorphic operations to systematically search over different ways of mapping input tensors into FHE vectors. For each choice, the compiler analyzes the resulting circuit to determine the encryption parameters that maximize performance while ensuring security and correctness. During this process, CHET additionally performs scheme-specific optimizations to increase the end-to-end performance.

Apart from the compiler, CHET includes a runtime, akin to the linear algebra libraries used in unencrypted evaluation of neural networks. We have developed a set of layouts and a unified metadata representation for them. For these new layouts, we have developed a set of computational kernels that implement the common operations found in convolutional neural networks (CNNs). All of these kernels were designed to use the vectorization capabilities of modern FHE schemes.

Schemes that support fixed-point arithmetic [15, 16] do so by scaling fixed-point numbers to integers. Determining the scaling factors to use for the inputs and the output is difficult as it involves a tradeoff between performance and output precision. CHET simplifies this choice with a profile-guided optimization step. Given a set of test inputs, CHET automatically determines the fixed-point scaling parameters that maximize performance while guaranteeing the desired precision requirements of the output for these test inputs.

We evaluate CHET with a set of real-world CNN models and show how different optimization choices available in our compiler can significantly improve inference latencies. As an example, for a neural network obtained from an industry partner for medical imaging, the base implementation took more than 18 hours per image, while a FHE expert was able to bring this to under 45 minutes with a hand-tuned circuit. By systematically searching over a wider set of possible optimizations, CHET generated a circuit that took less than 5 minutes per image. Moreover, CHET was able to easily port the same input circuit to a more recent and efficient FHE scheme that is harder to hand tune. Our port took less than a minute per image. CHET is also able to scale to large neural networks, such as SqueezeNet. To the best of our knowledge, this is the deepest neural network to be homomorphically evaluated.

The rest of this paper is organized as follows. Section 2 introduces homomorphic encryption and tensor programs. Section 3 provides an overview of using CHET with an example. Section 4 describes the intermediate representations and Section 5 describes our compiler. Section 6 presents our evaluation of CHET. Related work and conclusions are presented in Sections 7 and 8, respectively.

2 Background

This section provides background about FHE that is necessary to understand the contributions underlying CHET. Interested readers can look at [2] for more details.

2.1 Homomorphic Encryption

Say a plaintext message m is encrypted into a ciphertext $\langle m \rangle$ by an encryption scheme. This encryption scheme is said to be *homomorphic* with respect to an operation $\oplus$ if there exists an operation $\langle \oplus \rangle$ such that

$$\langle a \rangle \langle \oplus \rangle \langle b \rangle = \langle a \oplus b \rangle$$

for all messages a and b. For example, the popular encryption scheme RSA is homomorphic with respect to multiplication but not with respect to addition.

An encryption scheme is *fully homomorphic* (FHE) if it is homomorphic with respect to a set of operators that are sufficient to encode arbitrary computations. Current FHE schemes are *levelled* (also called as *somewhat homomorphic*) in that for fixed encryption parameters they only support computation of a particular depth.[1] In this paper, we will only deal with levelled homomorphic schemes as the size of the input circuit is known before hand.

2.2 Integer FHE Schemes with Rescaling

Early FHE schemes only supported Boolean operations. However, recent FHE schemes directly support integer *addition* and *multiplication* operations. This allows us to evaluate arbitrary arithmetic circuits efficiently without having to "explode" them into Boolean circuits. In addition, these FHE schemes also support optimized *constant* operations when one of the operand is a plaintext message.

Many applications, such as machine learning, require floating point arithmetic. To support such applications using integer FHE schemes, we can use fixed-point arithmetic with an explicit scaling factor. For instance, we can represent $\langle 3.14 \rangle$ as $\langle 314 \rangle$ with scale 100. However, this scaling factor quickly grows with multiplication. The resulting *plaintext-coefficient growth* [31] limits the size of the circuits one can practically evaluate. The recently proposed integer FHE scheme CKKS [16] enables *rescaling* ciphertexts — allowing one to convert say $\langle 2000 \rangle$ at fixed-point scale 100 to $\langle 20 \rangle$ at scale 1. This mitigates the problem of growing scaling factors. This paper only focusses on FHE schemes with rescaling support, namely the CKKS scheme [16] and its variant RNS-CKKS [15]. Obviously, CHET can trivially target other FHE schemes such as FV [21] or BGV [7], but this is not the main focus of the paper.

Note that many applications, including machine learning, use non-polynomial operations such as exp, log, and tanh. We will assume that such functions are appropriately approximated by polynomials before the circuit is provided to CHET. Prior work has already shown that this is feasible for machine learning [23].

[1] A levelled scheme may be turned into a fully homomorphic one by introducing a *bootstrapping* operation [22].

Table 1. The asymptotic costs of homomorphic operations for the CKKS and RNS-CKKS scheme variants in HEAAN v1.0 and SEAL v3.1 respectively. $M(Q)$ is the complexity of multiplying large integers and is $O(\log^{1.58} Q)$ for HEAAN.

Homomorphic Operation	CKKS	RNS-CKKS with $Q = \Pi_{i=1}^{r} Q_i$
addition, subtraction	$O(N \cdot \log Q)$	$O(N \cdot r)$
scalar multiplication	$O(N \cdot M(Q))$	$O(N \cdot r)$
plaintext multiplication	$O(N \cdot \log N \cdot M(Q))$	$O(N \cdot r)$
ciphertext multiplication	$O(N \cdot \log N \cdot M(Q))$	$O(N \cdot \log N \cdot r^2)$
ciphertext rotation	$O(N \cdot \log N \cdot M(Q))$	$O(N \cdot \log N \cdot r^2)$

2.3 Encryption Parameters

One of the essential tasks CHET performs is to automate the selection of encryption parameters that determine the correctness, security, and performance of the FHE computation. In both the CKKS and RNS-CKKS schemes, a ciphertext is a polynomial of degree N with each coefficient represented modulo Q. N is required to be a power of two in both schemes. While Q is a power of two in the CKKS scheme, Q is a product of r primes $Q = \Pi_{i=1}^{r} Q_i$ in the RNS-CKKS scheme. Table 1 shows the asymptotic cost of homomorphic operations for the CKKS and RNS-CKKS schemes implemented in HEAAN v1.0 [27] and SEAL v3.1 [38] respectively. Larger values of N and Q (or r) increase the cost of homomorphic operations. Note that although r can be much smaller than $\log(Q)$, real world performance should not be directly inferred from the asymptotic complexities, as implementations of the two schemes can have very different constants. Finally, the size of the encrypted messages grows with N.

While the performance constraints above require N and Q to be as small as possible, they have to be reasonably large for the following reasons. First, the parameter Q has to be large enough to correctly evaluate a given circuit. As the coefficients in the ciphertext polynomial are represented modulo Q, any coefficient that exceeds Q will result in an overflow, causing the message to be corrupted and unrecoverable. To avoid this overflow, the computation should periodically rescale the values as described in Section 2.2. But this rescaling "consumes" the modulus Q resulting in a polynomial with a smaller modulus. Thus, the computation has to perform a delicate balance between introducing sufficient rescaling to limit the growth of coefficient values and ensuring large-enough modulus to represent the output correctly. As multiplications are the primary reason for the growth of coefficients, Q directly limits the *multiplicative depth* of the circuit one can safely evaluate.

Additionally, for a given Q, larger values of N make the encryption harder to attack. The *security level* of an encryption is usually measured in bits, where n-bit security implies that a brute-force attack is expected to require at least 2^n operations. The security level for a given Q and N is a table

provided by the encryption scheme [12] which CHET explicitly encodes. By default, CHET chooses the smallest values of N and Q that guarantee 128-bit security.

2.4 FHE Vectorization

A unique capability of FHE schemes is the ability to support large *single instruction multiple data* (SIMD) vectors. These schemes, for appropriate setting of parameters, enable one to encode multiple integers into a larger integer and use the Chinese Remainder Theorem (see [31] for details) to simultaneously perform operations on individual integers by performing a single operation on the larger integer. When compared to SIMD capabilities of current hardware processors, these SIMD widths are large — vector sizes of tens of thousands or more are not uncommon. In particular, the SIMD width in CKKS and RNS-CKKS is $N/2$.

FHE schemes naturally support vector addition and multiplication. In addition, they support *rotation* operations that mimic the shuffle instructions of the SIMD units of modern processors. For instance, rotating a vector $[a_1, a_2, \ldots, a_n]$ by a constant i results in the vector $[a_{i+1}, \ldots, a_n, a_1, \ldots, a_i]$. However, FHE schemes do not support random access to extract a particular slot of a vector. Such operations need to be implemented by multiplying the vector with a plaintext mask followed by a rotation. This unfortunately adds to the multiplicative depth and should be avoided when possible.

Rotating a vector by an amount i requires a public rotation key that is specific to the constant i [16]. Given the large vector widths, it is impractical to generate a rotation key for every possible i. Instead, FHE libraries usually generate a public rotation key for every power-of-2 and then use multiple rotations to achieve the desired amount of rotation. CHET optimizes this by generating public rotation keys explicitly for a given input circuit.

2.5 Approximation in CKKS

As discussed above, FHE schemes introduce noise during homomorphic operations. Unlike other schemes, CKKS and RNS-CKKS are *approximate* and introduce noise in the lower-order bits of the message. To ensure that this noise does not affect the precision of the output, these schemes require that the inputs and the output be scaled by large-enough values. CHET includes a profile-guided optimization to determine these scaling factors automatically, given a set of test inputs.

2.6 Tensor Programs

A tensor is a multidimensional array with regular dimensions. A tensor t has a data type dtype(t), which defines the representation of each element, and a shape shape(t), which is a list of the tensor's dimensions. For example, a single 32 by 32 image with 3 channels of color values between 0 and 255 could be represented by a tensor I with dtype(I) = `int8` and shape(I) = [3, 32, 32].

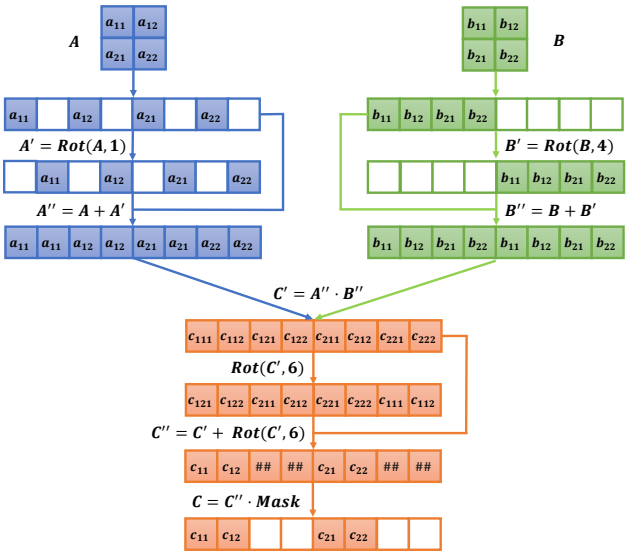

Figure 1. Homomorphic matrix-matrix multiplication.

In machine learning, neural networks are commonly expressed as programs operating on tensors. Some common tensor operations in neural networks include:

Convolution calculates a cross correlation between an input and a filter tensor.

Matrix multiplication represents neurons that are connected to all inputs, which are found in dense layers typically at the end of a convolutional neural network.

Pooling combines adjacent elements using a reduction operation such as maximum or average.

Element-wise operations such as batch normalization and ReLUs.

Reshaping reinterprets the shape of a tensor, for example, to flatten preceding a matrix multiplication.

We consider the tensor program as a circuit of tensor operations and this circuit is a Directed Acyclic Graph (DAG).

3 Overview of CHET

Developing an FHE application involves many challenges. CHET is an optimizing compiler designed to alleviate many of the issues. This section provides the overview of the compiler and how it aids in the development of FHE applications.

3.1 Motivating Example

We first motivate CHET with a simple example of homomorphically performing matrix-matrix multiplication. Figure 1 shows two 2×2 matrices A and B. We need to perform a total of 8 multiplications. The obvious way is to encrypt the scalar values of A and B individually and perform matrix-matrix multiplication the normal way. This is not a viable solution for large matrices due to the cost of FHE operations.

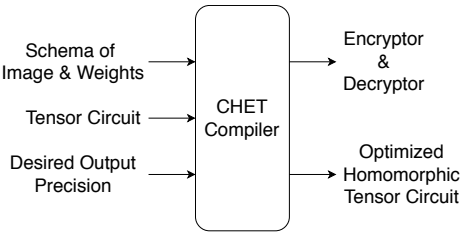

Figure 2. Overview of the CHET system at compile-time.

Instead, we would like to make use of the large SIMD capabilities provided by FHE schemes. This requires us to map the values onto vectors using specific *layouts*. In Figure 1, B is in the standard row-major format, but A's layout contains some padding. As shown in the figure, this special layout allows us to replicate the values twice, using rotations and additions, such that all the 8 products we need $c_{ijk} = a_{ij} \cdot b_{jk}$ can be obtained with 1 FHE multiplication in the C' vector.

We need additional rotation and addition operations to compute the elements of the output matrix $c_{ik} = \sum_j c_{ijk}$ in C''. This vector contains additional junk entries marked ##, which are masked out to produce the result C. Note that the layout of C is different from that of the input matrices A and B.

The key challenge here is that operations such as masking and rotations which are relatively cheap in plaintext SIMD operations are expensive in FHE. For instance, rotating C' by 6 would either require a special rotation key for 6 or we have to use two rotations using the power-of-2 keys for 4 and 2 generated by default by the underlying encryption schemes. Equally importantly, masking operations such as the one required to obtain C from C'' involve a multiplication that adds to the multiplicative depth.

Thus, when matrix C is used in a subsequent matrix multiplication, rather than converting it into a standard layout of A or B, we can emit a different set of instructions that are specific to the layout of C. Doing this manually while managing the different layouts of the variables in the program can soon become overwhelming and error prone.

While the simple example above already brought out the complexity of FHE programming, performing neural network inferencing brings out many more. First, we have to deal with multi-dimensional tensors where the choices of layouts are numerous. Moreover, useful machine learning models have large tensors that need not fit within the SIMD widths of the underlying schemes when using standard parameters. Thus, one has to deal with the trade-off between increasing the SIMD widths (by increasing the N parameter) or splitting tensors into multiple ciphertexts.

As we have seen above, layout decisions can change the operations one needs to perform which can in turn affect the multiplicative depth required to execute the circuit. However, setting the encryption parameters to allow the required multiplicative depth in turn changes the SIMD widths available which of course might require changes in the layout. Making these interdependent choices manually is a hard problem. CHET is designed to automatically explore these choices.

3.2 Using the Compiler

We will assume that the neural network inferencing computation is specified as a sequence of tensor operations that we call a tensor circuit. This input is very similar to how these models are specified in frameworks such as TensorFlow [1].

Figure 2 shows how an application programmer can use CHET to compile a tensor circuit. Consider a tensor circuit with a single operation:

$$output = conv2d(image, weights); \quad (1)$$

In addition to the tensor circuit, CHET requires the schema of the inputs to the circuit. The input schema specifies the tensor dimensions as well as the fixed-point scales to use for the tensor values. In Equation 1 for example, the user specifies that the input "image" (i) is encrypted, (ii) is a 4-dimensional tensor of size $1 \times 1 \times 28 \times 28$, and (iii) has a fixed-point scaling factor of 2^{40}. CHET also requires the desired fixed-point scales for the output (output precision).

Neural network models typically use floating-point values and determining the right fixed-point scaling factors to use is not straight-forward, especially given the approximation present in CKKS. Larger scaling factors might lead to higher cost of encrypted computation while smaller scaling factors might lead to loss in prediction accuracy. To aid in tuning the scaling factors, CHET provides an optional profile-guided optimization. Instead of specifying the scaling factors to use, the user provides a set of representative (training) images. CHET uses these to automatically determine appropriate fixed-point scales for the images, the weights, and the output.

For a target FHE scheme using the given constraints, CHET generates an equivalent, optimized homomorphic tensor circuit as well as an encryptor and decryptor. Both of these executables encode the choices made by the compiler to make the homomorphic computation efficient. For Equation 1, the homomorphic tensor circuit generated is:

$$encOutput = hconv2d(encImage, weights, HW); \quad (2)$$

There are several data layout options for the encrypted output of each operation and CHET chooses the best one; in this case, it chooses HW layout (described in Section 4.2). Similarly, the encryptor and decryptor use the encryption parameters decided by CHET. CHET also chooses the configuration of public keys that the encryptor should generate.

To evaluate the tensor circuit on an image, the client first generates a private key and encrypts the image using the encryptor generated by CHET, as shown in Figure 3. The encrypted image is then sent to the server along with unencrypted weights and public keys required for evaluating

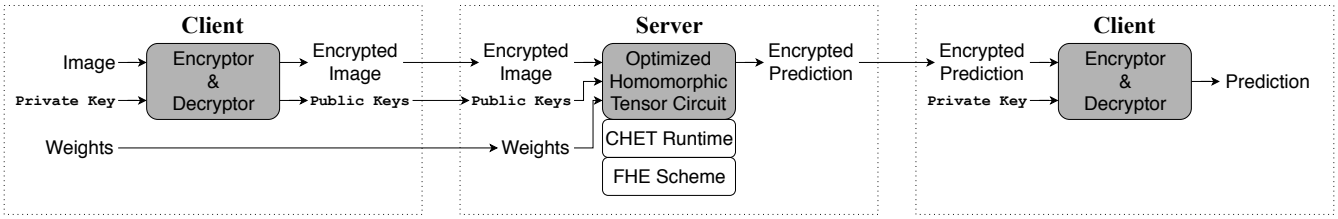

Figure 3. Overview of the CHET system at runtime (shaded boxes are programs generated by the CHET compiler).

homomorphic operations (i.e., multiplication and rotation). The server executes the optimized homomorphic tensor circuit generated by CHET. The homomorphic tensor operations in the circuit are executed using the CHET runtime, which uses the underlying target FHE scheme to execute homomorphic computations on encrypted data. The circuit produces an encrypted prediction, which it then sends to the client. The client decrypts the encrypted prediction with its private keys using the CHET generated decryptor. In this way, the client runs tensor programs like neural networks on the server without the server being privy to the data, the output (prediction), or any intermediate state.

In this paper, we assume a semi-honest threat model (like CryptoNets [23]) where the server and the compiler are semi-honest, i.e., the server and the compiler execute the requested computation faithfully but would be curious about the client or user data. The client or user data is private and its confidentiality must be guaranteed. As FHE is non-deterministic and immune to side-channel attacks, encrypting the user data or image using CHET is sufficient to ensure this. Note that CHET only knows the dimensions of the image and weights while the server only knows the image dimensions and the weights used in the model. CHET can be thus used by the client to offload both storage (encrypted data) and computation (neural network inference) to public cloud providers.

While Figure 3 presents the flow in CHET for homomorphically evaluating a tensor circuit on a single image, CHET supports evaluating the circuit on multiple images simultaneously, which is known as *batching* in image inference. However, unlike in neural network training, for inference tasks, it is not always true that a large batch-size is available. For example, in a medical imaging cloud service, where each patient's data is encrypted under their personal key, the batch is limited to the images from a single patient. While batching increases the throughput of image inference [23], CHET's focus is decreasing image inference latency. In the rest of this paper, we consider a batch size of 1, although CHET trivially supports larger batch sizes.

3.3 Design of the Compiler

A key design principle of CHET is the separation of concerns between the policies of choosing the secure, accurate, and most efficient homomorphic operation and the mechanisms of executing those policies. The policies include encryption parameters that determine that the computation is secure and accurate, and layout policies that map tensors onto vectors that are crucial for performance. Like Intel MKL libraries that have different implementations of linear algebra operations, the CHET runtime contains different implementations for tensor operations that cater to different input and output layouts. The CHET compiler is responsible for searching over the space of valid and efficient policies.

CHET introduces two abstractions that simplify its design and implementation. *Homomorphic Tensor Circuit* (HTC) is a tensor circuit annotated with the metadata that encodes all the policy decisions for every tensor variable. For instance in Figure 1, the HTC precisely specifies how the matrices A and B are mapped onto vectors and the desired layout for the output C. The HTC specification, including data layouts, is described in Section 4.2.

The CHET runtime implements the mechanisms necessary to support the policies in HTC. To design the runtime independent of the underlying FHE scheme, we introduce an abstraction called Homomorphic Instruction Set Architecture (HISA), in Section 4.1.

For each FHE scheme, the domain expert specifies the cost model of each HISA primitive in it (the models could be derived through theoretical or experimental analysis). These models use only local information (arguments of the HISA primitive) and are independent of the rest of the circuit. CHET uses these specifications to globally analyze the tensor circuit and choose the encryption parameters, the public keys configuration, and the data layouts. We describe these in detail in Section 5.

4 Intermediate Representations

CHET uses two intermediate representations: Homomorphic Instruction Set Architecture (HISA) and Homomorphic Tensor Circuit (HTC). HISA is a low-level intermediate representation, that acts as an interface between the CHET runtime and the underlying FHE scheme, whereas HTC is a high-level intermediate representation that acts as an interface between the CHET compiler and the CHET runtime. HISA and HTC are closely related to the operations in the target FHE library and the input tensor circuit, respectively.

Table 2. Primitives of the HISA.

Instruction	Semantics	Signature
encrypt(p)	Encrypt plaintext p into a ciphertext.	$\texttt{pt} \rightarrow \texttt{ct}$
decrypt(c)	Decrypt ciphertext c into a plaintext.	$\texttt{ct} \rightarrow \texttt{pt}$
copy(c)	Make a copy of ciphertext c.	$\texttt{ct} \rightarrow \texttt{ct}$
free(h)	Free any resources associated with handle h.	$\texttt{ct} \cup \texttt{pt} \rightarrow \texttt{void}$
encode(m, f)	Encode vector of reals m into a plaintext with a scaling factor f.	$\mathbb{R}^s, \mathbb{Z} \rightarrow \texttt{pt}$
decode(p)	Decode plaintext p into a vector of integers.	$\texttt{pt} \rightarrow \mathbb{R}^s$
rotLeft(c, x), rotLeftAssign(c, x)	Rotate ciphertext c left x slots.	$\texttt{ct}, \mathbb{Z} \rightarrow \texttt{ct}$
rotRight(c, x), rotRightAssign(c, x)	Rotate ciphertext c right x slots.	$\texttt{ct}, \mathbb{Z} \rightarrow \texttt{ct}$
add(c, c'), addAssign(c, c')		$\texttt{ct}, \texttt{ct} \rightarrow \texttt{ct}$
addPlain(c, p), addPlainAssign(c, p)	Add ciphertext, plaintext, or scalar to ciphertext c.	$\texttt{ct}, \texttt{pt} \rightarrow \texttt{ct}$
addScalar(c, x), addScalarAssign(c, x)		$\texttt{ct}, \mathbb{R} \rightarrow \texttt{ct}$
sub(c, c'), subAssign(c, c')		$\texttt{ct}, \texttt{ct} \rightarrow \texttt{ct}$
subPlain(c, p), subPlainAssign(c, p)	Subtract ciphertext, plaintext, or scalar from ciphertext c.	$\texttt{ct}, \texttt{pt} \rightarrow \texttt{ct}$
subScalar(c, x), subScalarAssign(c, x)		$\texttt{ct}, \mathbb{R} \rightarrow \texttt{ct}$
mul(c, c'), mulAssign(c, c')		$\texttt{ct}, \texttt{ct} \rightarrow \texttt{ct}$
mulPlain(c, p), mulPlainAssign(c, p)	Multiply ciphertext, plaintext, or scalar (at scale f) to ciphertext c.	$\texttt{ct}, \texttt{pt} \rightarrow \texttt{ct}$
mulScalar(c, x, f), mulScalarAssign(c, x, f)		$\texttt{ct}, \mathbb{R}, \mathbb{Z} \rightarrow \texttt{ct}$
rescale(c, x), rescaleAssign(c, x)	Rescale ciphertext c by scalar x. Undefined unless $\exists ub: x = \text{maxRescale}(c, ub)$.	$\texttt{ct}, \mathbb{Z} \rightarrow \texttt{ct}$
maxRescale(c, ub)	Returns the largest $d \leq ub$ that c can be rescaled by.	$\texttt{ct}, \mathbb{Z} \rightarrow \mathbb{Z}$

4.1 Homomorphic Instruction Set Architecture (HISA)

The goal of HISA is to abstract the details of FHE encryption schemes, such as the use of encryption keys and modulus management. This abstraction enables CHET to target new encryption schemes.

Table 2 presents instructions or primitives in the HISA. Each FHE library implementing the HISA provides two types: `pt` for plaintexts and `ct` for ciphertexts. During initialization, FHE library generates private keys required for encryption and decryption, and public keys required for evaluation. The appropriate use of these keys is the responsibility of the FHE library and is not exposed in HISA.

HISA supports point-wise fixed-point arithmetic operations and rotations. The number of slots in the FHE vector, s is a configurable parameter that is provided to the FHE library during the initialization. For schemes that do not support batching, this parameter can be set to 1. For such schemes, rotation operations are no-ops.

The rescale and maxRescale instructions abstract the rescaling operations provided by the CKKS [16] and the RNS-CKKS [15] schemes. These schemes have restrictions on the scalar value by which a ciphertext can be rescaled. For the CKKS scheme, the scalar has to be a power of 2. For the RNS-CKKS scheme, it has to be the next modulus in the modulus chain. The maxRescale(c, ub) abstracts this detail and returns the maximum value less than or equal to the desired ub that the input ciphertext c can be rescaled by. This instruction additionally guarantees that the return value is less than the modulus of c. For schemes that do not support rescaling, maxRescale always returns 1.

4.2 Homomorphic Tensor Circuit (HTC)

The goal of HTC is to provide a high-level abstraction of tensor operations and map them to low-level HISA primitives described above. This abstraction enables CHET to choose the most efficient layouts for input and intermediate tensors in the tensor circuit, and call the appropriate optimized runtime functions that implement tensor operations.

HTC provides an encrypted tensor datatype called CipherTensor. It contains metadata about the *logical layout* of the unencrypted tensor, such as its dimensions, padding information, and strides. The metadata is stored as plain integers as it does not leak any information about the data.

Since HISA only supports (1-dimensional) vectors, CipherTensor is responsible for mapping the tensor into its *physical layout* as a vector of ciphertexts, with each ciphertext encrypting a vector. This problem is similar to tiling or blocking of vectors to improve locality of high-performance kernels, but with different constraints. For example, consider a four-dimensional tensor commonly used in image recognition with batch (outermost) N, channel C, height H, and (innermost) width W dimensions. Figure 4 shows one way of mapping such a tensor A by representing each $H \times W$ matrix as a ciphertext in row-major format (with no padding and stride 1), and having $N \times C$ such ciphertexts in a vector. We call this the HW layout. One can also envision blocking the channel dimension too, where each ciphertext represents a $C \times H \times W$ tensor. We call this the CHW layout. We also support dividing the C dimension across multiple ciphertexts,

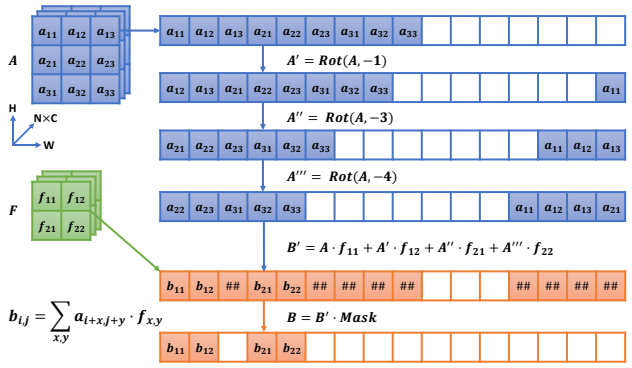

Figure 4. Homomorphic convolution of one channel in HW layout with a 2x2 filter (with valid padding).

in which case each ciphertexts represents a $c \times H \times W$ tensor for some $c < C$.

The metadata describing the physical layout of a CipherTensor also includes information about the strides for each dimension. For example, for an image of height (row) and width (column) of 28, a stride of 1 for the width dimension and a stride of 30 for the height dimension allows a padding of 2 (zero or invalid) elements between the rows.

It is not clear which layout is the most efficient way to represent a tensor. As hinted in Section 3, the physical layout of the tensors determines the instructions CHET needs to implement tensor operations. For instance, Figure 4 shows the convolution operation when the input image A and the filter F are in HW format. Convolution requires an element-wise multiplication of the image and the filter followed by an addition of neighboring elements. This can be implemented by rotating the image by appropriate amounts and multiplying each ciphertext with an element from the filter. When the values of the filter are unencrypted (which is the case when applying a known model to an encrypted image), this can be performed efficiently with a mulScalar.

Notice that the ciphertexts in Figure 4 have some empty space, which in many cases can be filled by using the CHW layout. However, in this case mulPlain needs to be used instead of mulScalar because different weights need to be multiplied to different 2-d HW images in the same ciphertext. As shown in Table 1, in RNS-CKKS, both mulPlain and mulScalar have $O(N \cdot r)$ complexity, whereas in CKKS, mulPlain and mulScalar have $O(N \cdot \log N \cdot M(Q))$ and $O(N \cdot M(Q))$ complexities, respectively. Therefore, for CKKS, homomorphic convolution in CHW layout is slower than that in HW layout, whereas for RNS-CKKS, CHW layout may be faster.

By abstracting the physical layout, HTC enables CHET to determine efficient layouts based on a global analysis of the entire circuit (described in Section 5). Moreover, by translating reshaping and padding operations in the circuit into operations that change the metadata, CHET avoids or delays performing these expensive operations only when necessary. HTC also enables us to incrementally support new tensor layouts. The current implementation of CHET only supports two layouts for each tensor: HW and CHW. We hope to extend CHET to support additional layouts in the future.

5 Compiling a Tensor Circuit

This section describes the CHET compiler. We first describe a global data-flow analysis and transformation framework (Section 5.1), which is instantiated for three different analyses to a) determine the encryption parameters (Section 5.2), b) choose an efficient layout for tensors (Section 5.3), and c) generate rotation keys necessary to evaluate the input circuit (Section 5.4). We then describe a profile-guided optimization to choose the fixed-point scales for the inputs and output (Section 5.5).

5.1 Analysis and Transformation Framework

The input tensor circuit is first translated to an equivalent Homomorphic Tensor Circuit (HTC) as follows: (i) any tensor marked as encrypted in the input schema is considered to be of type CipherTensor, and (ii) any tensor operation with at least one input CipherTensor is mapped to an equivalent homomorphic tensor operation that produces a CipherTensor as output. CHET then analyses and transforms the HTC.

Analyzing the HTC involves performing a sequence of data flow analyses on the HTC. Each such analysis requires an FHE expert to specify the data-flow equation for each HISA primitive. We call the set of these equations the HISA-Analyser. Each equation reads the data-flow information of its inputs and writes the data-flow information of its output, and is agnostic of the rest of the circuit.

The obvious way to run the data-flow analysis is to explicitly build the data-flow graph of HISA primitives from the HTC. In contrast to most traditional optimizing compilers, the HTC has two key properties: its data-flow graph is a Directed Acyclic Graph (DAG) and the dimensions of tensors in the graph are known at compile-time from the schema provided by the user (similar to High Performance Fortran compilers). CHET exploits this to elide building such a data-flow graph in memory.

Instead, CHET performs the analysis by dynamically unrolling the graph *on-the-fly*. This is done by executing the HTC and the CHET runtime using a different interpretation. In our implementation, we customize the ct datatype in HISA primitives using templates to store data-flow information and overload operations to execute the data-flow equations rather than call into the underlying FHE library. Thus, executing the framework in this new interpretation automatically composes the data-flow equations to perform the analysis. Once a HISA-Analyser has finished, a subsequent *transformer* uses the data-flow information computed to transform the circuit appropriately.

To summarize, a compiler transformation pass in CHET for a particular FHE scheme can be defined by the following: (1) ct *datatype* to store the data-flow information along with its initialization, (2) data-flow equation for each HISA primitive (*HISA-Analyser*), and (3) a *transformer* that takes the data-flow analysis results and returns a specification for the HTC. The framework handles the rest. The analysis and transformation framework can thus be used to easily add new transformations and new target FHE schemes.

5.2 Encryption Parameters Selection

The goal of this analysis is to determine the encryption parameters to use when evaluating a given circuit. These parameters are also used in the Encryptor and Decryptor to encrypt the inputs and decrypt the results respectfully. As described in Section 2.3, parameters N and Q determine the performance, security, and correctness of an FHE computation in the CKKS and RNS-CKKS schemes.

For a given Q, the minimum value of N that guarantees that security level against currently known attacks is a deterministic map, as listed in [12] (Section 5.4). We pre-populate this in a table and choose 128-bit security (this is a configurable hyper-parameter). The data-flow analysis in this compiler pass is thus to aid in determining only the coefficient modulus, Q.

As described in Section 2.3, maximizing performance requires us to find a tight lower-bound for Q that is sufficiently large to evaluate the input circuit correctly and to the desired precision. In both CKKS and RNS-CKKS, the coefficient modulus changes during execution of the rescale instruction: rescale takes the ciphertext message c with modulus m and an integer x, and produces a ciphertext c' with modulus $m' = m/x$; in effect, the modulus is "consumed" in the rescale operation. If the new modulus is below 1, then the resulting ciphertext is invalid. Hence, tracking the modulus consumed using data-flow analysis can find the minimum required Q.

The challenge in tracking the modulus consumed is that both CKKS and RNS-CKKS restrict the divisors that can be used in the rescale instruction (the maxRescale instruction can be used to query a suitable divisor for a given bound). To analyze their behavior at runtime, we use the analysis framework to execute the instructions using a different interpretation. This interpretation tracks how the modulus changes with rescale (Section 2.3) and which divisors are valid, as returned from maxRescale. Using this, the encryption parameters selection pass for CKKS is defined by:

Datatype ct stores the modulus "consumed" and is initialized to 1.

HISA-Analyser The maxRescale instruction returns the same value that it would have returned if it was executed in CKKS. The rescale instruction multiplies the divisor with the input's ct, and stores it to the output's ct, thereby tracking the modulus consumed during computation. In all other instructions, if the output is a ct, then it is the same as the inputs' ct (which are required to all be the same).

Transformer The circuit output's ct captures the modulus "consumed" during the entire circuit. The user-provided desired output fixed-point scaling factor (precision) is multiplied by the circuit output's ct to get Q. Finally, N is chosen using the pre-populated table.

For the RNS-CKKS scheme, the analysis assumes there is a global list $Q_1, Q_2, \ldots, Q_n$ of pre-generated candidate moduli[2] for a sufficiently large n. The goal of the analysis is to pick the smallest r such that $Q = \Pi_{i=1}^{r} Q_i$ can be used as the modulus for the input circuit.

Datatype ct stores the index k in the list of moduli to be "consumed" next and is initialized to 1.

HISA-Analyser For the maxRescale(c, ub) and rescale instructions, the analyser determines the largest $j \geq k$ such that $\Pi_{i=k}^{j} Q_i \leq ub$. If such a j exists, maxRescale returns $\Pi_{i=k}^{j} Q_i$ and rescale stores the index $j+1$ to the output's ct. If no such j exists, rescale stores the index k and maxRescale returns 1. In all other instructions, if the output is a ct, then it is the same as the inputs' ct (which are required to all be the same).

Transformer The circuit output's ct captures the number of moduli "consumed" during the entire circuit. The length of the modulus chain is then chosen as the smallest r such that $\Pi_{i=ct}^{r} Q_i$ is greater than the user-provided desired output fixed-point scaling factor (precision) multiplied by the circuit output's ct. Q is set to $\Pi_{i=1}^{r} Q_i$. Finally, N is chosen using the pre-populated table.

5.3 Data Layout Selection

The goal of the data layout selection pass is to determine the data layout of the output of each homomorphic tensor operation in the HTC so that the execution time of the HTC is minimized. We first need to search the space of data layout choices. For each such choice, we then need to analyze the cost of the HTC corresponding to that choice. To do this, we need an estimation of the cost of each HISA primitive. We describe each of these three components in detail next.

The cost of HISA primitives could be estimated using theoretical analysis (through asymptotic complexity) or experimental analysis (through microbenchmarking). Table 1 lists the asymptotic complexity of different HISA primitives in CKKS and RNS-CKKS. Different HISA primitives have different costs, even within the same FHE scheme. In this paper, we use a combination of theoretical and experimental analysis, by using asymptotic complexity and tuning the constants involved using microbenchmarking of CKKS and

[2] By default, CHET uses a list of 60-bit primes distributed in SEAL.

RNS-CKKS instructions, to derive a cost model for HISA primitives (agnostic of the tensor circuit and inputs).

Given a cost model for HISA primitives, data-flow analysis is used to estimate the cost of executing a HTC with a specific data layout choice. We define the cost estimation pass for both CKKS and RNS-CKKS as follows:

Datatype ct stores the cost and is initialized to 0.
HISA-Analyser For each HISA primitive, if the output is a ciphertext, then its ct is the sum of the ct of the inputs and the cost of that primitive, according to the cost model specific to CKKS and RNS-CKKS.
Transformer The circuit output's ct captures the cost of the entire HTC. If the cost is smaller than the minimum observed so far, then the minimum cost is updated and the best data layout choice is set to the current one.

Note that as we execute HISA instructions using this new interpretation, we use parallel threads during analysis to estimate the cost on each thread and take the maximum across threads as the cost of the HTC.

The runtime exposes the data layout choices for the outputs of homomorphic tensor operations. In our current runtime (more details can be found in [19]), there are only 2 such choices per tensor operation, HW and CHW layouts (Section 4.2). The search-space for the HTC is exponential in the number of tensor operations, so it is huge. An auto-tuner might be useful to explore this search space. We instead use domain-specific heuristics to prune this search space: (i) homomorphic convolutions are typically faster if the input and output are in HW, while all the other homomorphic tensor operations are typically faster if the input and output are in CHW, and (ii) homomorphic matrix multiplications (fully connected layers) are typically faster when the output is in CHW, even if the input is in HW, while all other homomorphic tensor operations are typically faster if both input and output are in the same layout. Using these heuristics, we consider only 4 data layout choices for the HTC: (i) HW: all homomorphic tensor operations use HW, (ii) CHW: all homomorphic tensor operations use CHW, (iii) HW-conv, CHW-rest: homomorphic convolution uses HW, while all other homomorphic tensor operations use CHW, and (iv) CHW-fc, HW-before: all homomorphic tensor operations till the first homomorphic fully connected layer use HW and everything thereafter uses CHW. We thus restrict the search space to a constant size. Each data layout choice corresponds to a HTC. For each choice, we select the encryption parameters and analyze the cost (two passes). We then pick the minimum cost one.

5.4 Rotation Keys Selection

The goal of the rotation keys selection pass is to determine the public evaluation keys for rotation that needs to be generated by the encryptor and decryptor. In both CKKS and RNS-CKKS, to rotate a ciphertext by x slots, a public evaluation key for rotating by x is needed. Recall that the vector width used for FHE vectorization is $N/2$. Since the range of possible rotations of this vector, i.e., $N/2$, is huge and each rotation key consumes significant memory, it is not feasible to generate rotation keys for each possible rotation. Therefore, both CKKS and RNS-CKKS, by default, insert public evaluation keys for power-of-2 left and right rotations, and all rotations are performed using a combination of power-of-2 rotations. As a consequence, any rotation that is not a power-of-2 would perform worse because it has to rotate multiple times. Nevertheless, only $2log(N) - 2$ rotation keys are stored by default. This is too conservative. In a given homomorphic tensor circuit, the distinct slots to rotate would not be in the order of N. We use data-flow analysis to track the distinct slots of rotations used in the HTC.

We define the rotation keys selection pass for both CKKS and RNS-CKKS as follows:

Datatype ct stores the set of rotation slots used and is initialized to $\emptyset$.
HISA-Analyser Any rotate HISA instruction inserts the slots to rotate to the input's ct, and stores it to the output's ct. In all other instructions, if the output is a ciphertext, then its ct is the union of the inputs' ct.
Transformer The circuit output's ct captures all the rotation slots used in the HTC. Rotation keys for these slots are marked for generation.

5.5 Fixed-Point Scaling Factor Selection

The inputs to the compiler passes discussed so far include the fixed-point scales of the inputs (images and weights) in HTC as well as the desired fixed-point scale (or precision) of the output. It is not straightforward for the user to determine the scaling factors to use for the floating-point values. Moreover, these factors interact with the approximation noise introduced by the FHE scheme, so the fixed-point scales must be large enough to ensure that the noise added does not reduce accuracy too much. To alleviate this, CHET includes an optional profile-guided optimization that finds appropriate scaling factors. Instead of fixed-point scales, the user provides unencrypted inputs along with a tolerance (or error-bound) for the output.

Floating-point numbers are encoded as fixed-point numbers by multiplying them by their specified fixed-point scale (inputs are scaled before encryption if applicable). Recall that multiplication of two such fixed-pointed numbers results in a number at a larger fixed-point scale, that may be *rescaled* back down if a suitable divisor exists (as provided by maxRescale). The magnitude of the divisors (for CKKS) or when divisors become available (for RNS-CKKS) is determined by the fixed-point scales of the operands. Thus, the selection of scaling factors directly impacts rescale and consequently, encryption parameters selection.

Table 3. Deep Neural Networks used in our evaluation.

Network	No. of layers			# FP operations	Accuracy (%)
	Conv	FC	Act		
LeNet-5-small	2	2	4	159960	98.5
LeNet-5-medium	2	2	4	5791168	99.0
LeNet-5-large	2	2	4	21385674	99.3
Industrial	5	2	6	-	-
SqueezeNet-CIFAR	10	0	9	37759754	81.5

Table 4. Encryption parameters N and Q selected by CHET-HEAAN and the user-provided fixed-point parameters.

Network	N	$\log(Q)$	$\log(P_c,$	$P_w,$	$P_u,$	$P_m)$
LeNet-5-small	8192	240	30	16	15	8
LeNet-5-medium	8192	240	30	16	15	8
LeNet-5-large	16384	400	40	20	20	10
Industrial	32768	705	35	25	20	10
SqueezeNet-CIFAR	32768	940	30	20	20	10

Larger scaling factors yield larger encryption parameters and worse performance, whereas smaller scaling factors yield smaller encryption parameters and better performance but the output might vary beyond the tolerance, leading to prediction inaccuracy. The compiler minimizes the scaling factors while ensuring that the output values are within tolerance for the given inputs.

Given a set of scaling factors, we use CHET to generate the optimized HTC. For each input, we encrypt it using parameters chosen by CHET, run the HTC, and decrypt the output. We compare this with the output of the unencrypted tensor circuit. If the difference is beyond tolerance for any of the output values for any input, then we reject these scaling factors. Otherwise, the scaling factors are acceptable.

For neural network inference, CHET allows specifying 4 fixed-point scaling factors - one for the image, one for masks, and two for the weights depending on whether they are used as scalars or plaintext (vectors)[3]. If there are x choices for selecting the scaling factors for each, then there are x^4 choices in total. To limit the search space to $4 \cdot x$, we used a round-robin strategy: each scaling factor starts from 2^{40} and we decrease the exponents by one as long as the accuracy is acceptable. The search continues until a minimum is reached.

6 Evaluation

We evaluate the CHET compiler with two target FHE libraries: HEAAN v1.0 [27] and SEAL v3.1 [38], that implement the FHE schemes, CKKS [16] and RNS-CKKS [15], respectively. Our evaluation targets a set of convolutional neural network (CNN) architectures for image classification tasks that are summarized in Table 3.

LeNet-5-like is a series of networks for the MNIST [33] dataset. We use three versions with different number of neurons: LeNet-5-small, LeNet-5-medium, and LeNet-5-large. The largest one matches the one used in the TensorFlow's tutorials [39]. These networks have two convolutional layers, each followed by ReLU activation and max pooling, and two fully connected layers with a ReLU in between.

Industrial is a pre-trained HE-compatible neural network from an industry partner for privacy-sensitive binary classification of images. We are unable to reveal the details of the network other than the fact that it has 5 convolutional layers and 2 fully connected layers.

SqueezeNet-CIFAR is a neural network for the CIFAR-10 dataset [30] that follows the SqueezeNet [26] architecture. This version has 4 Fire-modules [18] for a total of 10 convolutional layers. To the best of our knowledge, SqueezeNet-CIFAR is the deepest NN that has been homomorphically evaluated.

All networks other than Industrial use ReLUs and max-pooling, which are not compatible with homomorphic evaluation (HE). For these networks, we modified the activation functions to a second-degree polynomial [11, 23]. The key difference with prior work is that our activation functions are $f(x) = ax^2 + bx$ with learnable parameters a and b. During the training phase, the CNN adjusts these parameters to implement an appropriate activation function. We also replaced max-pooling with average-pooling.

Table 3 lists the accuracies for the HE-compatible networks. For LeNet-5-large, the 99.3% accuracy matches that of the unmodified network. For SqueezeNet-CIFAR, the resulting accuracy of 81.5% is close to that of the non-HE-compatible network, which achieves an accuracy of 84%. Note that encryption was not used during training. The learned weights are used for inference with an encrypted image.

To provide a fair comparison with a set of hand-written HEAAN baselines that used non-standard encryption parameters, experiments with CHET-HEAAN were run with matching parameters that offer somewhat less than 128-bit security. Experiments with CHET-SEAL use the default 128-bit security level.

All experiments were run on a dual socket Intel Xeon E5-2667v3@3.2GHz with 224 GB of memory. Hyperthreading was turned off for a total of 16 hardware threads. We present the average latency of image inference with a batch size of 1. All latencies are reported as averages over 20 different images.

Comparison with hand-written: For a fair comparison, we consider hand-written implementations using HEAAN for a subset of the networks. The RNS-CKKS scheme in SEAL is difficult to manually hand tune, and thus we do not compare with hand-written version for SEAL. Experts

[3] Separate scaling factors are used because approximation noise of encoding in CKKS is smaller when all the elements in a plaintext are equal.

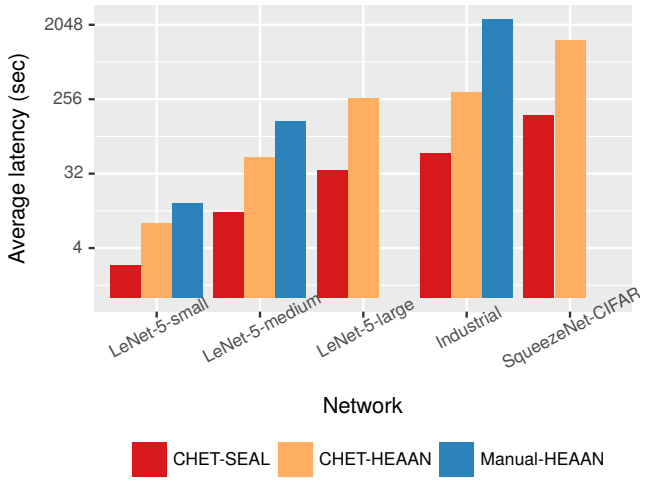

Figure 5. Average latency (log scale) of CHET-SEAL, CHET-HEAAN, and hand-written HEAAN versions.

Table 5. Average latency (sec) with different data layouts using CHET-SEAL.

Network	HW	CHW	HW-conv CHW-rest	CHW-fc HW-before
LeNet-5-small	**2.5**	3.8	3.8	**2.5**
LeNet-5-medium	22.1	**10.8**	25.8	18.1
LeNet-5-large	64.8	**35.2**	64.6	61.2
Industrial	108.4	**56.4**	181.1	136.3
SqueezeNet-CIFAR	429.3	**164.7**	517.0	441.0

Table 6. Average latency (sec) with different data layouts using CHET-HEAAN.

Network	HW	CHW	HW-conv CHW-rest	CHW-fc HW-before
LeNet-5-small	8	12	8	**8**
LeNet-5-medium	82	91	52	**51**
LeNet-5-large	325	423	270	**265**
Industrial	330	**312**	379	381
SqueezeNet-CIFAR	**1342**	1620	1550	1342

took weeks of programming effort to optimize each network independently. They reduced the latency of LeNet-5-small and LeNet-5-medium to 14 and 140 seconds, respectively. For Industrial, the default implementation by a developer took more than 18 hours per image, which the experts were able to tune to less than 45 minutes after months of work.

Figure 5 compares the hand-written implementations with CHET generated optimized ones for HEAAN and SEAL. For context, CHET-SEAL is around two orders of magnitude slower than CHET's unencrypted reference inference engine. CHET clearly outperforms hand-written implementations, even when using HEAAN. The hand-written implementations are slower because it is tedious and error prone

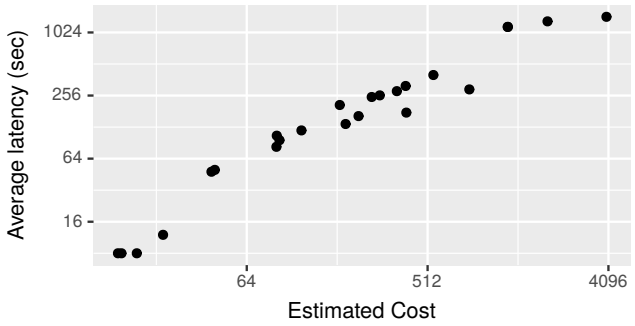

Figure 6. Estimated cost vs. observed average latency (log-log scale) for different layouts and networks in CHET.

to explore different data layouts. CHET not only explores layouts but also chooses the best one automatically. Furthermore, CHET-SEAL is an order of magnitude faster than hand-written implementations, while the complications of the RNS-CKKS scheme are automatically taken care of.

Cryptonets [23] homomorphically evaluate a highly tuned neural network for the MNIST [33] data using the YASHE [5] scheme. While this neural network is smaller than our LeNet-5-small, its accuracy is similar to that of our LeNet-5-medium. The average latency of image inference in their highly optimized implementation is 250 seconds (throughput is higher because they consider a larger batch size).

Parameter Selection: The encryption parameters N and Q selected by CHET are shown in Table 4 for HEAAN with the best data layout (SEAL is omitted due to lack of space). The values of these parameters grow with the depth of the circuit. The last columns show the fixed-point scaling parameters that were used for the image (P_c), plaintext (P_w) and scalar weights (P_u), and masks (P_m). With these parameters, encrypted inference achieved the same accuracy as unencrypted inference of the same HE-compatible networks.

Data Layout Selection: It is time-consuming to evaluate all possible data layouts, so we evaluate only the pruned subset that CHET searches by default. The pruned data layouts that we evaluate are: (i) HW: each ciphertext has all height and width elements of a single channel only, (ii) CHW: each ciphertext can have multiple channels (all height and width elements of each), (iii) HW-conv and CHW-rest: same has CHW, but move to HW before each convolution and back to CHW after each convolution, and (iv) CHW-fc and HW-before: same as HW, but switch to CHW during the first fully connected layer and CHW thereafter. Tables 5 and 6 present the average latency of different layouts for SEAL and HEAAN. The best data layout depends on the network as well as the FHE scheme. For example, the time for convolutions in HEAAN could be more for CHW layout than HW layout because mulPlain is more time-consuming than mulScalar. On the other hand, mulPlain and mulScalar take similar time

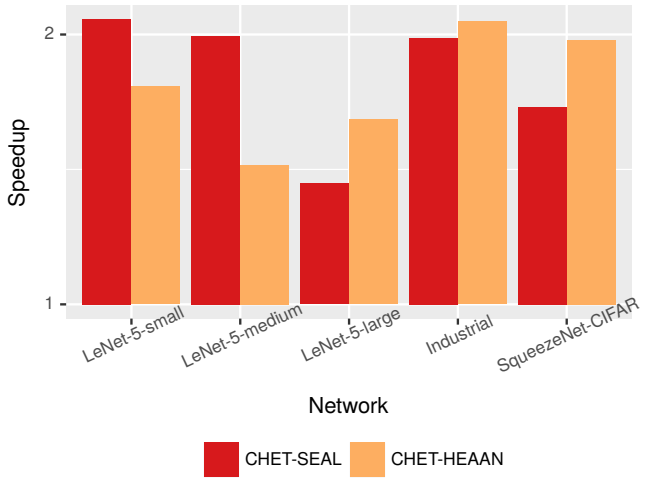

Figure 7. Speedup (log scale) of rotation keys selection optimization over default power-of-2 rotation keys.

in SEAL. Hence, for the same network, CHW layout might be the best layout in SEAL but not in HEAAN. It is very difficult for the user to determine which is the best data layout and it is tedious to implement each network manually using a different data layout. For all these networks, CHET chooses the best performing data layout automatically based on its cost model of SEAL or HEAAN. Figure 6 plots the cost estimated by CHET and the observed average latency for each network and data layout. We observe that they are highly correlated, so our cost model is quite accurate.

Rotation Keys Selection: Figure 7 presents the speedup of using only CHET selected rotation keys over using power-of-2 rotation keys (by default). The geometric mean speedup is 1.8× for all networks and FHE schemes. We observe that the rotation keys chosen by the compiler are a constant factor of $log(N)$ in every case. CHET thus significantly improves performance without consuming noticeably more memory.

7 Related Work

FHE is currently an active area of research [3–5, 7, 8, 13–17, 21, 22, 32, 41]. See Acar et al [2] for a survey of FHE schemes. CHET can be used to homomorphically evaluate tensor programs using any of these schemes. We demonstrated the utilities of CHET for the most recent FHE schemes, CKKS [16] and its RNS variant [15].

Many have observed the need and suitability of FHE for machine learning tasks [11, 23, 25, 28]. Cryptonets [23] was the first tool to demonstrate a fully-homomorphic inference of a CNN by replacing the (FHE incompatible) ReLU activations with a quadratic function. Our emphasis in this paper is primarily on automating the manual and error-prone hand tuning required to ensure that the networks are secure, correct, and efficient.

Bourse et al. [6] use the TFHE library [17] for CNN inference. TFHE operates on bits and is thus slow for multi-bit integer arithmetic. To overcome this difficulty, Bourse et al. instead use a discrete binary neural network. Similarly, Cingulata [9, 10] is a compiler for converting C++ programs into a Boolean circuit, which is then evaluated using a backend FHE library. Despite various optimizations [9], these approaches are unlikely to scale for large CNNs.

Prior works [20, 40] have used partially homomorphic encryption schemes (which support addition or multiplication of encrypted data but not both) to determine the encryption schemes to use for different data items so as to execute a given program. While they are able to use computationally efficient schemes, such techniques are not applicable for evaluating CNNs that require both multiplication and addition to be done on the same input.

DeepSecure [37] and Chameleon [36] use secure multiparty computation techniques [24, 42] to perform CNN inference. Many frameworks like SecureML [35], MiniONN [34], and Gazelle [29] combine secure two-party computation protocols [42] with homomorphic encryption. Secure multiparty computations require the client and the server to collaboratively perform computation, involving more communication and more computation on the client when compared to FHE. These tradeoffs between FHE and other secure computation techniques are beyond the scope of the paper. Our focus is instead on optimizing FHE computations. Nevertheless, a compiler like CHET could possibly be useful in other secure computing techniques as well.

8 Conclusions

Recent cryptographic breakthroughs have pushed FHE into the realm of practical applications. This paper shows that an optimizing compiler can outperform expert-hand-tuned implementations by systematically exploring many more optimization choices than manually feasible. In addition, CHET makes it easy to port the same input circuits to different FHE schemes, a task that will become necessary as new and improved FHE schemes are discovered. While our focus in this paper has been on encoding tensor circuits for homomorphic neural-network inference, an application that is of utmost importance for us, we believe that the techniques described in this paper can be generalized to other domains and applications. We also believe that with continued cryptographic innovations, the task of developing practical FHE applications will become a "systems and compiler" problem, requiring foundational research from the PL community.

Acknowledgments

We thank the anonymous reviewers and in particular our shepherd, Martin Hirzel, for their many suggestions in improving this paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2017. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *CoRR* abs/1704.03578 (2017). arXiv:1704.03578 http://arxiv.org/abs/1704.03578

[3] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. 2017. A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In *Selected Areas in Cryptography – SAC 2016*, Roberto Avanzi and Howard Heys (Eds.). Springer, 423–442. https://doi.org/10.1007/978-3-319-69453-5_23

[4] Fabrice Benhamouda, Tancrède Lepoint, Claire Mathieu, and Hang Zhou. 2017. Optimization of Bootstrapping in Circuits. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2423–2433. http://dl.acm.org/citation.cfm?id=3039686.3039846

[5] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. 2013. Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme. In *Cryptography and Coding*, Martijn Stam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 45–64. https://doi.org/10.1007/978-3-642-45239-0_4

[6] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 483–512. https://doi.org/10.1007/978-3-319-96878-0_17

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 13. https://doi.org/10.1145/2090236.2090262

[8] Z. Brakerski and V. Vaikuntanathan. 2014. Efficient Fully Homomorphic Encryption from (Standard) LWE. *SIAM J. Comput.* 43, 2 (2014), 831–871. https://doi.org/10.1137/120868669 arXiv:https://doi.org/10.1137/120868669

[9] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. 2018. A Multi-start Heuristic for Multiplicative Depth Minimization of Boolean Circuits. In *Combinatorial Algorithms*, Ljiljana Brankovic, Joe Ryan, and William F. Smyth (Eds.). Springer International Publishing, 275–286. https://doi.org/10.1007/978-3-319-78825-8_23

[10] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing (SCC '15)*. ACM, New York, NY, USA, 13–19. https://doi.org/10.1145/2732516.2732520

[11] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. 2017. Privacy-Preserving Classification on Deep Neural Network. *IACR Cryptology ePrint Archive* (2017), 35. https://eprint.iacr.org/2017/035/20170216:192421

[12] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. [n. d.]. Security of Homomorphic Encryption. http://homomorphicencryption.org/white_papers/security_homomorphic_encryption_white_paper.pdf.

[13] Hao Chen. 2017. Optimizing relinearization in circuits for homomorphic encryption. *CoRR* abs/1711.06319 (2017). https://arxiv.org/abs/1711.06319.

[14] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 360–384. https://doi.org/10.1007/978-3-319-78381-9_14

[15] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A Full RNS variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2018*. Springer. https://doi.org/10.1007/978-3-030-10970-7_16 LNCS 11349.

[16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017 (LNCS 10624)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15.

[17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. Cryptology ePrint Archive, Report 2018/421. https://eprint.iacr.org/2018/421.

[18] David Corvoysier. 2017. SqueezeNet for CIFAR-10. https://github.com/kaizouman/tensorsandbox/tree/master/cifar10/models/squeeze.

[19] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2018. CHET: Compiler and Runtime for Homomorphic Evaluation of Tensor Programs. *CoRR* abs/1810.00845 (2018). arXiv:1810.00845 http://arxiv.org/abs/1810.00845

[20] Yao Dong, Ana Milanova, and Julian Dolby. 2016. JCrypt: Towards Computation over Encrypted Data. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. 8:1–8:12. https://doi.org/10.1145/2972206.2972209

[21] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* (2012), 144. https://eprint.iacr.org/2012/144

[22] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[23] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 201–210. http://jmlr.org/proceedings/papers/v48/gilad-bachrach16.html

[24] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[25] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. 2017. CryptoDL: Deep Neural Networks over Encrypted Data. *CoRR* abs/1711.05189 (2017). arXiv:1711.05189 http://arxiv.org/abs/1711.05189

[26] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). https://arxiv.org/abs/1602.07360.

[27] Cryptography Lab in Seoul National University. [n. d.]. Homomorphic Encryption for Arithmetic of Approximate Numbers (HEAAN). https://github.com/snucrypto/HEAAN.

[28] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. 2018. Secure Outsourced Matrix Computation and Application to Neural Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1209–1222. https://doi.org/10.1145/3243734.3243837

[29] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1651–1669. https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[30] Alex Krizhevsky. 2009. The CIFAR-10 Dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[31] Kim Laine. 2017. Simple Encrypted Arithmetic Library (SEAL) Manual. https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf.

[32] Kristin Lauter. 2017. Postquantum Opportunities: Lattices, Homomorphic Encryption, and Supersingular Isogeny Graphs. *IEEE Security Privacy* 15, 4 (2017), 22–27. https://doi.org/10.1109/MSP.2017.3151338

[33] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. [n. d.]. The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/.

[34] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 619–631. https://doi.org/10.1145/3133956.3134056

[35] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. 19–38. https://doi.org/10.1109/SP.2017.12

[36] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18)*. ACM, New York, NY, USA, 707–721. https://doi.org/10.1145/3196494.3196522

[37] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. 2017. DeepSecure: Scalable Provably-Secure Deep Learning. *CoRR* abs/1705.08963 (2017). arXiv:1705.08963 http://arxiv.org/abs/1705.08963

[38] SEAL 2019. Microsoft SEAL 3.1.0. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[39] TensorFlow 2016. LeNet-5-like convolutional MNIST model example. https://github.com/tensorflow/models/blob/v1.9.0/tutorials/image/mnist/convolutional.py.

[40] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd D. Millstein. 2013. MrCrypt: Static Analysis for Secure Cloud Computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 271–286. https://doi.org/10.1145/2509136.2509554

[41] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology – EUROCRYPT 2010*, Henri Gilbert (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–43. https://doi.org/10.1007/978-3-642-13190-5_2

[42] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. IEEE Computer Society, Washington, DC, USA, 162–167. https://doi.org/10.1109/SFCS.1986.25