RESEARCH-ARTICLE

EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation

ROSHAN DATHATHRI, The University of Texas at Austin, Austin, TX, United States

BLAGOVESTA KOSTOVA, EPFL, Lausanne, Switzerland

OLLI SAARIKIVI, Microsoft Research, Redmond, WA, United States

WEI DAI, Microsoft Research, Redmond, WA, United States

KIM LAINE, Microsoft Research, Redmond, WA, United States

MADANLAL MUSUVATHI, Microsoft Research, Redmond, WA, United States

Open Access Support provided by:

The University of Texas at Austin

Microsoft Research

EPFL

Published: 11 June 2020

Citation in BibTeX format

PLDI '20: 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation
June 15 - 20, 2020
London, UK

Conference Sponsors:
SIGPLAN

EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation

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

Blagovesta Kostova
EPFL, Switzerland
blagovesta.pirelli@epfl.ch

Olli Saarikivi
Microsoft Research, USA
olsaarik@microsoft.com

Wei Dai
Microsoft Research, USA
wei.dai@microsoft.com

Kim Laine
Microsoft Research, USA
kilai@microsoft.com

Madan Musuvathi
Microsoft Research, USA
madanm@microsoft.com

Abstract

Fully-Homomorphic Encryption (FHE) offers powerful capabilities by enabling secure offloading of both storage and computation, and recent innovations in schemes and implementations have made it all the more attractive. At the same time, FHE is notoriously hard to use with a very constrained programming model, a very unusual performance profile, and many cryptographic constraints. Existing compilers for FHE either target simpler but less efficient FHE schemes or only support specific domains where they can rely on expert-provided high-level runtimes to hide complications.

This paper presents a new FHE language called Encrypted Vector Arithmetic (EVA), which includes an optimizing compiler that generates correct and secure FHE programs, while hiding all the complexities of the target FHE scheme. Bolstered by our optimizing compiler, programmers can develop efficient general-purpose FHE applications directly in EVA. For example, we have developed image processing applications using EVA, with a very few lines of code.

EVA is designed to also work as an intermediate representation that can be a target for compiling higher-level domain-specific languages. To demonstrate this, we have re-targeted CHET, an existing domain-specific compiler for neural network inference, onto EVA. Due to the novel optimizations in EVA, its programs are on average 5.3× faster than those generated by CHET. We believe that EVA would enable a wider adoption of FHE by making it easier to develop FHE applications and domain-specific FHE compilers.

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

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

ACM Reference Format:
Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386023

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386023

1 Introduction

Fully-Homomorphic Encryption (FHE) allows arbitrary computations on encrypted data without requiring the decryption key. Thus, FHE enables interesting privacy-preserving capabilities, such as offloading secure storage and secure computation to untrusted cloud providers. Recent advances in FHE theory [12, 13] along with improved implementations have pushed FHE into the realm of practicality. For instance, with appropriate optimization, we can perform encrypted fixed-point multiplications within a few microseconds, which matches the speed of 8086 processors that jumpstarted the computing revolution. Future cryptographic innovations will further reduce the performance gap between encrypted and unencrypted computations.

Despite the availability of multiple open-source implementations [25, 28, 38, 41], programming FHE applications remains hard and requires cryptographic expertise, making it inaccessible to most programmers today. Furthermore, different FHE schemes provide subtly different functionalities and require manually setting encryption parameters that control correctness, performance, and security. We expect the programming complexity to only increase as future FHE schemes become more capable and performant. For instance, the recently invented CKKS scheme [13] supports fixed-point arithmetic operations by representing real numbers as integers with a fixed scaling factor, but requires the programmer to perform rescaling operations so that scaling factors and the cryptographic noise do not grow exponentially due

to multiplications. Moreover, the so-called RNS-variant of the CKKS scheme [12] provides efficient implementations that can use machine-sized integer operations as opposed to multi-precision libraries, but imposes further restrictions on the circuits that can be evaluated on encrypted data.

To improve the developer friendliness of FHE, this paper proposes a new general-purpose language for FHE computation called Encrypted Vector Arithmetic (EVA). EVA is also designed to be an intermediate representation that is a backend for other domain-specific compilers. At its core, EVA supports arithmetic on fixed-width vectors and scalars. The vector instructions naturally match the encrypted SIMD – or batching – capabilities of FHE schemes today. EVA includes an optimizing compiler that hides all the complexities of the target FHE scheme, such as encryption parameters and noise. It ensures that the generated FHE program is correct, performant, and secure. In particular, it eliminates all common runtime exceptions that arise when using FHE libraries.

EVA implements FHE-specific optimizations, such as optimally inserting operations like rescaling and modulus switching. We have built a compiler incorporating all these optimizations to generate efficient programs that run using the Microsoft SEAL [41] FHE library which implements the RNS-variant of the CKKS scheme. We have built an EVA executor that transparently parallelizes the generated program efficiently, allowing programs to scale well. The executor also automatically reuses the memory used for encrypted messages, thereby reducing the memory consumed.

To demonstrate EVA's usability, we have built a Python frontend for it. Using this frontend, we have implemented several applications in EVA with a very few lines of code and much less complexity than in SEAL directly. One application computes the length of a path in 3-dimensional space, which can be used in secure fitness mobile applications. We have implemented some statistical machine learning applications. We have also implemented two image processing applications, Sobel filter detection and Harris corner detection. We believe Harris corner detection is one of the most complex programs that have been evaluated using CKKS.

In addition, we have built a domain-specific compiler on top of EVA for deep neural network (DNN) inference. This compiler takes programs written in a higher-level language as input and generates EVA programs using a library of operations on higher-level constructs like tensors and images. In particular, our DNN compiler subsumes the recently proposed domain-specific compiler called CHET [18]. Our DNN compiler uses the same tensor kernels as CHET, except that it generates EVA programs instead of generating SEAL programs. Nevertheless, the optimizing compiler in EVA is able to outperform CHET in DNN inference by 5.3× on average.

In summary, EVA is a general-purpose language and an intermediate representation that improves the programmability of FHE applications by guaranteeing correctness and security, while outperforming current methods.

The rest of this paper is organized as follows. Section 2 gives background on FHE. Section 3 presents the EVA language. Section 4 gives an overview of the EVA compiler. We then describe transformations and analysis in the compiler in Sections 5 and 6 respectively. Section 7 briefly describes the domain-specific compilers we built on top of EVA. Our evaluation is presented in Section 8. Finally, related work and conclusions are presented in Sections 9 and 10 respectively.

2 Background and Motivation

In this section, we describe FHE (Section 2.1) and the challenges in using it (Section 2.2). We also describe an implementation of FHE (Section 2.3). Finally, we present the threat model assumed in this paper (Section 2.4).

2.1 Fully-Homomorphic Encryption (FHE)

An FHE scheme includes four stages: key generation, encryption, evaluation, and decryption. Most of the efficient FHE schemes, for example, BGV [6], BFV [19], and CKKS [13], are constructed on the Ring Learning with Errors (RLWE) problem [34]. At the time of key generation, a polynomial ring of degree N with integer coefficients modulo Q must be chosen to represent ciphertexts and public keys according to the security standard [1]. We call Q the ciphertext modulus. A message is encoded to a polynomial, and subsequently encrypted with a public key or a secret key to form a ciphertext consisting of two polynomials of degree up to $N-1$. Encryption also adds to a ciphertext a small random error that is later removable in decryption.

FHE schemes are malleable by design. From the perspective of the user, they offer a way to encrypt integers (or fixed-point numbers in CKKS — see the next section) such that certain arithmetic operations can be evaluated on the resulting ciphertexts. Evaluation primarily includes four operations: addition of ciphertexts, addition of a ciphertext and a plaintext, multiplication of ciphertexts, and multiplication of a ciphertext and a plaintext. Decrypting (with a secret key) and decoding reveals the message, as if the computation was performed on unencrypted data.

Many modern FHE schemes also include a SIMD-like feature known as *batching* which allows a vector of values to be encrypted as a single ciphertext ($N/2$ values in CKKS). With batching, arithmetic operations happen in an element-wise fashion. *Batching-compatible* schemes can evaluate rotations which allow data movement inside a ciphertext. But evaluating each rotation step count needs a distinct public key.

2.2 Challenges in Using FHE

Programmers using FHE face significant challenges that must be overcome for correct, efficient, and secure computation. We discuss those challenges here to motivate our work.

Depth of Computation: Computations on ciphertexts increase the initially small error in them linearly on the

number of homomorphic additions and exponentially on the multiplicative depth of the evaluation circuit. When the errors get too large, ciphertexts become corrupted and cannot be decrypted, even with the correct secret key. This bound is in turn determined by the size of the encryption parameter Q. Thus, to support efficient homomorphic evaluation of a circuit, one must optimize the circuit for low depth.

Relinearization: Each ciphertext consists of 2 or more polynomials (freshly encrypted ciphertexts consist of only 2 polynomials). Multiplication of two ciphertexts with k and l polynomials yields a ciphertext with $k + l + 1$ polynomials. To prevent the number of polynomials from growing indefinitely, an operation called relinearization is performed to reduce it back to 2. Relinearization is costly and their optimal placement is an NP-hard problem [10].

CKKS and Approximate Fixed-Point: The CKKS [13] scheme introduced an additional challenge by only providing *approximate results* (but much higher performance in return). There are two main sources of error in CKKS: (i) error from the encoding of values to polynomials being lossy, and (ii) the noise added in every homomorphic operation being mixed with the message. To counter this, CKKS adopts a fixed-point representation by associating each ciphertext with an unencrypted scaling factor. Using high enough scaling factors allows the errors to be hidden.

CKKS further features an operation called *rescaling* that scales down the fixed-point representation of a ciphertext. Consider a ciphertext x that contains the encoding of 0.25 multiplied by the scale 2^{10} (a relatively low scale). x^2 encodes 0.0625 multiplied by the scale 2^{20}. Further powers would rapidly overflow modest values of the modulus Q, requiring impractically large encryption parameters to be selected. Rescaling the second power by 2^{10} will truncate the fixed-point representation to encode the value at a scale of 2^{10}.

Rescaling has a secondary effect of also dividing the ciphertext's modulus Q by the same divisor as the ciphertext itself. This means that there is a limited "budget" for rescaling built into the initial value of Q. The combined effect for CKKS is that $\log Q$ can grow linearly with the multiplicative depth of the circuit. It is common to talk about the *level* of a ciphertext as how much Q is left for rescaling.

A further complication arises from the ciphertext after rescaling being encrypted under fundamentally different encryption parameters. To apply any binary homomorphic operations, two ciphertexts must be at the same level, i.e., have the same Q. Furthermore, addition and subtraction require ciphertexts to be encoded at the same scale due to the properties of fixed-point arithmetic. CKKS also supports a modulus switching operation to bring down the level of a ciphertext without scaling the message. *In our experience, inserting the appropriate rescaling and modulus switching operations to match levels and scales is a significantly difficult process even for experts in homomorphic encryption.*

Table 1. Types of values.

Type	Description
Cipher	An encrypted vector of fixed-point values.
Vector	A vector of 64-bit floating point values.
Scalar	A 64-bit floating point value.
Integer	A 32-bit signed integer.

In the most efficient implementations of CKKS (so called RNS-variants [11]), the truncation is actually performed by dividing the encrypted values by prime factors of Q. Furthermore, there is a fixed order to these prime factors, which means that from a given level (i.e., how many prime factors are left in Q) there is only one valid divisor available for rescaling. This complicates selecting points to rescale, as doing so too early might make the fixed-point representation so small that the approximation errors destroy the message.

Encryption Parameters: In CKKS, all of the concerns about scaling factors, rescaling, and managing levels are intricately linked with selecting encryption parameters. Thus, a typical workflow when developing FHE applications involves a lot of trial-and-error, and repeatedly tweaking the parameters to achieve both correctness (accuracy) and performance. While some FHE libraries warn the user if the selected encryption parameters are secure, but not all of them do, so a developer may need to keep in mind security-related limitations, which typically means upper-bounding Q for a given N.

2.3 Microsoft SEAL

Microsoft SEAL [41] is a software library that implements the RNS variant of the CKKS scheme. In SEAL, the modulus Q is a product of several prime factors of bit sizes up to 60 bits, and rescaling of ciphertexts is always done by dividing away these prime factors. The developer must choose these prime factors and order them correctly to achieve the desired rescaling behavior. SEAL automatically validates encryption parameters for correctness and security.

2.4 Threat Model

We assume a semi-honest threat model, as is typical for homomorphic encryption. This means that the party performing the computation (i.e., the server) is curious about the encrypted data but is guaranteed to run the desired operations faithfully. This model matches for example the scenario where the server is trusted, but a malicious party has read access to the server's internal state and/or communication between the server and the client.

3 EVA Language

The EVA framework uses a single language as its input format, intermediate representation, and executable format. The

Table 2. Instruction opcodes and their semantics (see Section 2 for details on semantics of the restricted instructions).

Opcode	Signature	Description	Restrictions
Negate	**Cipher** → **Cipher**	Negate each element of the argument.	
Add	**Cipher** × (**Vector**|**Cipher**) → **Cipher**	Add arguments element-wise.	
Sub	**Cipher** × (**Vector**|**Cipher**) → **Cipher**	Subtract right argument from left one element-wise.	
Multiply	**Cipher** × (**Vector**|**Cipher**) → **Cipher**	Multiply arguments element-wise (and multiply scales).	
RotateLeft	**Cipher** × **Integer** → **Cipher**	Rotate elements to the left by given number of indices.	
RotateRight	**Cipher** × **Integer** → **Cipher**	Rotate elements to the right by given number of indices.	
Relinearize	**Cipher** → **Cipher**	Apply relinearization.	Not in input
ModSwitch	**Cipher** → **Cipher**	Switch to the next modulus in the modulus chain.	Not in input
Rescale	**Cipher** × **Scalar** → **Cipher**	Rescale the ciphertext (and divide scale) with the scalar.	Not in input

EVA language abstracts *batching-compatible* FHE schemes like BFV [19], BGV [6], and CKKS [12, 13], and can be compiled to target libraries implementing those schemes. Input programs use a subset of the language that omits details specific to FHE, such as when to rescale. In this section, we describe the input language and its semantics, while Section 4 presents an overview of the compilation to an executable EVA program.

Table 1 lists the types that values in EVA programs may have. The vector types **Cipher** and **Vector** have a fixed power-of-two size for each input program. The power-of-two requirement comes from the target encryption schemes.

We introduce some notation for talking about types and values in EVA. For **Vector**, a literal value with elements a_i is written $[a_1, a_2, \ldots, a_i]$ or as a comprehension $[a_i \text{ for } i = 1 \ldots i]$. For the ith element of **Vector** a, we write a_i. For the product type (i.e., tuple) of two EVA types A and B, we write $A \times B$, and write tuple literals as (a, b) where $a \in A$ and $b \in B$.

Programs in EVA are Directed Acyclic Graphs (DAGs), where each node represents a value available during execution. Example programs are shown in Figures 1(a) and 2(a). Nodes with one or more incoming edges are called *instructions*, which compute a new value as a function of its *parameter* nodes, i.e., the parent nodes connected to it. For the ith parameter of an instruction n, we write $n.parm_i$ and the whole list of parameter nodes is $n.parms$. Each instruction n has an opcode $n.op$, which specifies the operation to be performed at the node. Note that the incoming edges are ordered, as it corresponds to the list of arguments. Table 2 lists all the opcodes available in EVA. The first group are opcodes that frontends may generate, while the second group lists FHE-specific opcodes that are inserted by the compiler. The key to the expressiveness of the input language are the RotateLeft and RotateRight instructions, which abstract rotation (circular shift) in *batching-compatible* FHE schemes.

A node with no incoming edges is called a *constant* if its value is available at compile time and an *input* if its value is only available at run time. For a constant n, we write $n.value$ to denote the value. Inputs may be of any type, while constants can be any type except **Cipher**. This difference is due to the fact that the **Cipher** type is not fully defined before key generation time, and thus cannot have any values at compile time. The type is accessible as $n.type$.

A program P is a tuple $(M, Insts, Consts, Inputs, Outputs)$, where M is the length of all vector types in P; $Insts$, $Consts$ and $Inputs$ are list of all instruction, constant, and input nodes, respectively; and $Outputs$ identifies a list of instruction nodes as outputs of the program.

Next, we define execution semantics for EVA. Consider a dummy encryption scheme *id* that instead of encrypting **Cipher** values just stores them as **Vector** values. In other words, the encryption and decryption are the identity function. This scheme makes homomorphic computation very easy, as every plaintext operation is its own homomorphic counterpart. Given a map $I : Inputs \to \textbf{Vector}$, let $\mathcal{E}_{id}(n)$ be the function that computes the value for node n recursively by using $n.value$ or $I(n)$ if n is a constant or input respectively and using $n.op$ and $\mathcal{E}_{id}()$ on $n.parms$ otherwise. Now for a program P, we further define its reference semantic as a function P_{id}, which given a value for each input node maps each output node in P to its resulting value:

$$P_{id} : \times_{n_i \in Inputs} n_i.type \to \times_{n_o \in Outputs} \textbf{Vector}$$
$$P_{id}(I(n_i^1), \ldots, I(n_i^{|Inputs|})) = (\mathcal{E}_{id}(n_o^1), \ldots, \mathcal{E}_{id}(n_o^{|Outputs|}))$$

These execution semantics hold for any encryption scheme, except that output is also encrypted (i.e., **Cipher** type).

The EVA language has a serialized format defined using Protocol Buffers [23], a language and platform neutral data serialization format. Additionally, the EVA language has an in-memory graph representation that is designed for efficient analysis and transformation, which is discussed in Section 4.

4 Overview of EVA Compiler

In this section, we describe how to use the EVA compiler (Section 4.1). We then describe the constraints on the code

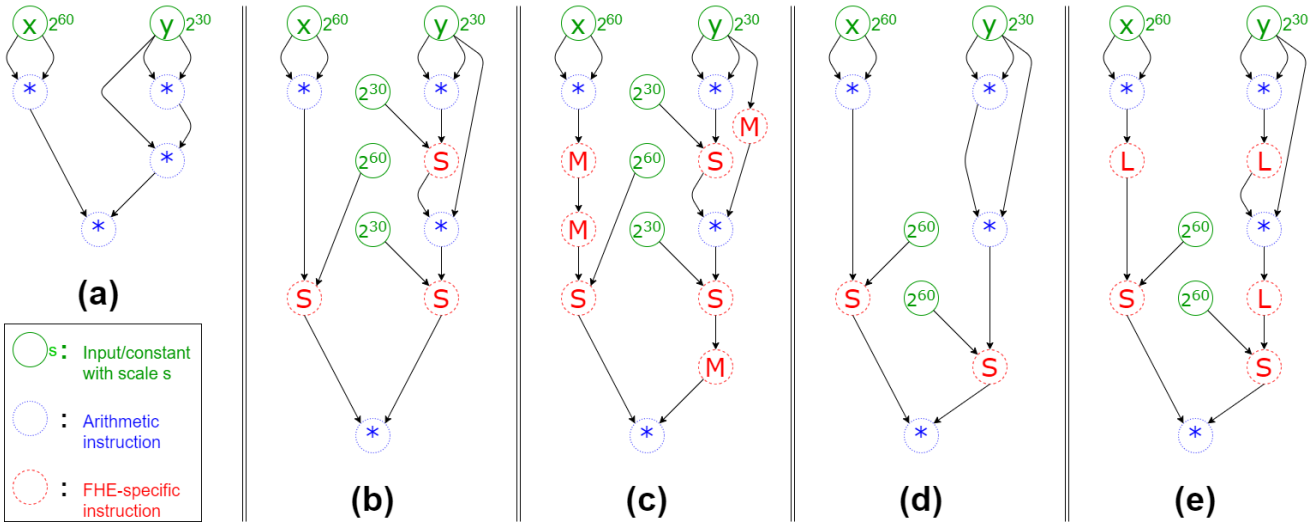

Figure 1. x^2y^3 example in EVA: (a) input; (b) after ALWAYS-RESCALE; (c) after ALWAYS-RESCALE & MODSWITCH; (d) after WATERLINE-RESCALE; (e) after WATERLINE-RESCALE & RELINEARIZE (S: RESCALE, M: MODSWITCH, L: RELINEARIZE).

generated by EVA (Section 4.2). Finally, we give an overview of the execution flow of the EVA compiler (Section 4.3).

4.1 Using the Compiler

In this paper, we present the EVA compiler for the RNS variant of the CKKS scheme [12] and its implementation in the SEAL library [41]. Targeting EVA for other FHE libraries [25, 28, 38] implementing CKKS [12, 13] would be straightforward. The EVA compiler can also be adapted to support other *batching-compatible* FHE schemes like BFV [19] and BGV [6].

The EVA compiler takes a program in the EVA language as input. Along with the program, it needs the fixed-point scales or precisions for each input in the program and the desired fixed-point scales or precisions for each output in the program. The compiler then generates a program in the EVA language as output. In addition, it generates a vector of bit sizes that must be used to generate the encryption parameters as well as a set of rotation steps that must be used to generate the rotation keys. The encryption parameters and the rotations keys thus generated are required to execute the generated EVA program.

While the input and the output programs are in the EVA language, the set of instructions allowed in the input and the output are distinct, as listed in Table 2. The RELINEARIZE, RESCALE, and MODSWITCH instructions require understanding the intricate details of the FHE scheme. Hence, they are omitted from the input program.

The input scales and the desired output scales affect the encryption parameters, and consequently, the performance and accuracy of the generated program. Choosing the right values for these is a trade-off between performance and accuracy (while providing the same security). Larger values lead to larger encryption parameters and more accurate but slower generated program, whereas smaller values lead to smaller encryption parameters and less accurate but faster generated program. Profiling techniques like those used in prior work [18] can be used to select the appropriate values.

4.2 Motivation and Constraints

There is a one-to-one mapping between instructions in the EVA language (Table 2) and instructions in the RNS-CKKS scheme. However, the input program cannot be directly executed. Firstly, encryption parameters are required to ensure that the program would be accurate. EVA can simply determine the bit sizes that is required to generate the parameters. However, this is insufficient to execute the program correctly because some instructions in the RNS-CKKS scheme have restrictions on their inputs. If these restrictions are not met, the instructions would just throw an exception at runtime.

Each ciphertext in the RNS-CKKS scheme has a coefficient modulus q ($Q = \prod_{i=1}^{r} q_i$)[1] and a fixed-point *scale* associated with it. All freshly encrypted ciphertexts have the same q but they may have different *scale*. The following constraints apply for the binary instructions involving two ciphertexts:

$$n.parm_1.modulus = n.parm_2.modulus \quad \text{if } n.op \in \{\text{ADD}, \text{SUB}, \text{MULTIPLY}\} \tag{1}$$

$$n.parm_1.scale = n.parm_2.scale \quad \text{if } n.op \in \{\text{ADD}, \text{SUB}\} \tag{2}$$

In the rest of this paper, whenever we mention ADD regarding constraints, it includes both ADD and SUB.

[1] In SEAL, if the coefficient modulus q is $\{q_1, q_2, \ldots, q_r\}$, then q_i is a prime close to a power-of-2. EVA compiler (and the rest of this paper) assumes q_i is the corresponding power-of-2 instead. To resolve this discrepancy, when a RESCALE instruction divides the scale by the prime, the scale is adjusted (by the EVA executor) as if it was divided by the power-of-2 instead.

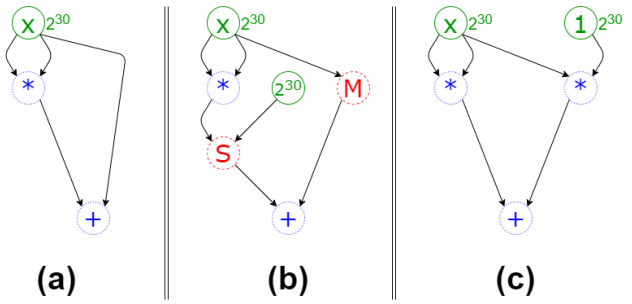

Figure 2. x^2+x example in EVA: (a) input; (b) after ALWAYS-RESCALE & MODSWITCH; (c) after MATCH-SCALE.

Consider the example to compute x^2y^3 for ciphertexts x and y in Figure 1 (viewed as a dataflow graph). Constraint 2 is trivially satisfied because they are no ADD instructions. Only RESCALE and MODSWITCH instructions modify q. Constraint 1 is also trivially satisfied due to the absence of these instructions. Nonetheless, without the use of RESCALE instructions, the scales and the *noise* of the ciphertexts would grow exponentially with the multiplicative depth of the program (i.e., maximum number of MULTIPLY nodes in any path) and consequently, the $\log_2$ of the coefficient modulus product Q required for the input would grow exponentially. Instead, using RESCALE instructions ensures that $\log_2 Q$ would only grow linearly with the multiplicative depth of the program.

In Figure 1, the output has a scale of $2^{60} * 2^{60} * 2^{30} * 2^{30} * 2^{30}$ (with $x.scale = 2^{60}$ and $y.scale = 2^{30}$). This would require Q to be at least $2^{210} * s_o$, where s_o is the user-provided desired output scale. To try to reduce this, one can insert RESCALE after every MULTIPLY, as shown in Figure 1(b). However, this yields an invalid program because it violates Constraint 1 for the last (bottom) MULTIPLY (and there is no way to choose the same q for both x and y). To satisfy this constraint, MODSWITCH instructions can be inserted, as shown in Figure 1(c). Both RESCALE and MODSWITCH drop the first element in their input q (or *consume the modulus*), whereas RESCALE also divides the scale by the given scalar (which is required to match the first element in q). The output now has a scale of $2^{60} * 2^{30}$. This would require choosing $q = \{2^{30}, 2^{30}, 2^{60}, 2^{60}, 2^{30}, s_o\}$. Thus, although Figure 1(c) executes more instructions than Figure 1(a), it requires the same Q. A better way to insert RESCALE instructions is shown in Figure 1(d). This satisfies Constraint 1 without MODSWITCH instructions. The output now has a scale of $2^{60} * 2^{30}$. We can choose $q = \{2^{60}, 2^{60}, 2^{30}, s_o\}$, so $Q = 2^{150} * s_o$. Hence, this program is more efficient than the input program.

If the computation was modified to $x^2 + y^3$ in Figure 1(d), then the last (bottom) MULTIPLY would be replaced by ADD and the program would violate Constraint 2 as ADD would have operands with scales 2^{60} and 2^{30}. Consider a similar but simpler example of $x^2 + x$ in Figure 2(a). One way to satisfy Constraints 1 and 2 is by adding RESCALE and MODSWITCH, as shown in Figure 2(b), which would require $q = \{2^{30}, 2^{30}, s_o\}$. Another way is to introduce MULTIPLY of x and a constant 1 with 2^{30} scale to match the scale of ADD operands, as shown in Figure 2(c), which would require $q = \{2^{60}, s_o\}$. Although the product $Q = 2^{60} * s_o$ is same in both cases, the modulus length r is different. Hence, the program in Figure 2(c) is more efficient due to a smaller r.

MULTIPLY has another constraint. Each ciphertext consists of 2 or more polynomials. MULTIPLY of two ciphertexts with k and l polynomials yields a ciphertext with $k + l + 1$ polynomials. Nevertheless, fewer the polynomials faster the MULTIPLY, so we enforce it to be the minimum:

$$\begin{gathered}\forall i\ n.parm_i.num\_polynomials = 2 \\ \text{if } n.op \in \{\text{MULTIPLY}\}\end{gathered} \tag{3}$$

RELINEARIZE reduces the number of polynomials in a ciphertext to 2. It can be inserted in the program in Figure 1(d) to satisfy this constraint, as shown in Figure 1(e).

Finally, we use s_f to denote the maximum allowed rescale value in the rest of this paper ($\log_2 s_f$ is also the maximum bit size that can be used for encryption parameters), i.e.,

$$\begin{gathered}n.parm_2.value \le s_f \\ \text{if } n.op \in \{\text{RESCALE}\}\end{gathered} \tag{4}$$

In the SEAL library, $s_f = 2^{60}$ (which enables a performant implementation by limiting scales to machine-sized integers).

To summarize, FHE schemes (or libraries) are tedious for a programmer to reason about, due to all their cryptographic constraints. Programmers find it even more tricky to satisfy the constraints in a way that optimizes performance. *The EVA compiler hides such cryptographic details from the programmer while optimizing the program.*

4.3 Execution Flow of the Compiler

As mentioned in Section 3, the in-memory internal representation of the EVA compiler is an **Abstract Semantic Graph**, also known as a **Term Graph**, of the input program. In the rest of this paper, we will use the term *graph* to denote an Abstract Semantic Graph. In this in-memory representation, each node can access both its parents and its children, and for each output, a distinct leaf node is added as a child. It is straightforward to construct the graph from the EVA program and vice-versa, so we omit the details. We use the terms program and graph interchangeably in the rest of the paper.

Algorithm 1 presents the execution flow of the compiler. There are four main steps, namely transformation, validation, parameters selection, and rotations selection. The transformation step takes the input program and modifies it to satisfy the constraints of all instructions, while optimizing it. In the next step, the transformed program is validated to ensure that no constraints are violated. If any constraints are violated, then the compiler throws an exception. By doing this, EVA ensures that executing the output program will never lead to a runtime exception thrown by the FHE library. Finally, for

Algorithm 1: Execution of EVA compiler.

Input : Program P_i in EVA language
Input : Scales S_i for inputs in P_i
Input : Desired scales S_d for outputs in P_i
Output: Program P_o in EVA language
Output: Vector B_v of bit sizes
Output: Set R_s of rotation steps

P_o = **Transform**(P_i, S_i)
if ***Validate***(P_o) == *Failed* **then**
  Throw an exception
B_v = **DetermineParameters**(P_o, S_i, S_d)
R_s = **DetermineRotationSteps**(P_i, S_i)

the validated output program, the compiler selects the bit sizes and the rotation steps that must be used to determine the encryption parameters and the rotation keys respectively, before executing the output program. The transformation step involves rewriting the graph, which is described in detail in Section 5. The other steps only involve traversal of the graph (without changing it), which is described in Section 6.

5 Transformations in EVA Compiler

In this section, we describe the key graph transformations in the EVA compiler. We first describe a general graph rewriting framework (Section 5.1). Then, we describe the graph transformation passes (Sections 5.2 and 5.3).

5.1 Graph Rewriting Framework

A graph transformation can be captured succinctly using graph *rewrite rules* (or term rewrite rules). These rules specify the conditional transformation of a subgraph (or an expression) and the graph transformation consists of transforming all applicable subgraphs (or expressions) in the graph (or program). The graph nodes have read-only properties like the opcode and number of parents. In a graph transformation, some state or data may be stored on each node in the graph and the rewrite rules may read and update the state.

The rewrite rules specify local operations on a graph and the graph transformation itself is composed of applying these operations wherever needed. The schedule in which these local operations are applied may impact the correctness or efficiency of the transformation. Consider two schedules: (1) forward pass from roots to leaves of the graph (a node is scheduled for rewriting only after all its parents have already been rewritten), or (2) backward pass from leaves to roots of the graph (a node is scheduled for rewriting only after all its children have already been rewritten). Note that the rewriting operation may not do any modifications if its condition does not hold. In forward pass, state (or data) flows from parents to children. Similarly, in backward pass, state (or data) flows from children to parents. In general, multiple forward or backward passes may be needed to apply the rewrite rules until quiescence (no change).

EVA includes a graph rewriting framework for arbitrary rewrite rules for a subgraph that consists of a node along with its parents or children. The rewrite rules for each graph transformation pass in EVA are defined in Figure 3. A single backward pass is sufficient for EAGER-MODSWITCH, while a single forward pass is sufficient for the rest. The rewrite rules assume the passes are applied in a specific order: WATERLINE-RESCALE, EAGER-MODSWITCH, MATCH-SCALE, and RELINEARIZE (ALWAYS-RESCALE and LAZY-MODSWITCH are not used but defined only for clarity). For the sake of exposition, we will first describe RELINEARIZE pass before describing the other passes.

5.2 Relinearize Insertion Pass

Each ciphertext is represented as 2 or more polynomials. Multiplying two ciphertexts each with 2 polynomials yields a ciphertext with 3 polynomials. The Relinearize instruction reduces a ciphertext to 2 polynomials. To satisfy Constraint 3, EVA must insert Relinearize after Multiply of two nodes with **Cipher** type and before another such Multiply.

The RELINEARIZE rewrite rule (Figure 3) is applied for a node n only if it is a Multiply operation and if both its parents (or parameters) have **Cipher** type. The transformation in the rule inserts a Relinearize node n_l between the node n and its children. In other words, the new children of n will be only n_l and the children of n_l will be the old children of n. For the example graph in Figure 1(d), applying this rewrite rule transforms the graph into the one in Figure 1(e).

Optimal placement of relinearization is an NP-hard problem [10]. Our relinearization insertion pass is a simple way to enforce Constraint 3. More advanced relinearization insertion, with or without Constraint 3, is left for future work.

5.3 Rescale and ModSwitch Insertion Passes

Goal: The Rescale and ModSwitch nodes (or instructions) must be inserted such that they satisfy Constraint 1, so the goal of the Rescale and ModSwitch insertion passes is to insert them such that the coefficient moduli of the parents of any Add and Multiply node are equal.

While satisfying Constraint 1 is sufficient for correctness, performance depends on where Rescale and ModSwitch are inserted (as illustrated in Section 4.2). Different choices lead to different coefficient modulus $q = \{q_1, q_2, ..., q_r\}$, and consequently, different polynomial modulus N for the roots (or inputs) to the graph (or program). Larger values of N and r increase the cost of every FHE operation and the memory of every ciphertext. N is a non-decreasing function of $Q = \prod_{i=1}^{r} q_i$ (i.e., if Q grows, N either remains the same or grows as well). Minimizing both Q and r is a hard problem to solve. However, reducing Q is only impactful if it reduces N, which is unlikely as the threshold of Q, for which N increases, grows exponentially. Therefore, *the goal of EVA is to yield the optimal r*, which may or may not yield the optimal N.

$$\text{ALWAYS} - \text{RESCALE}\ \frac{n \in \mathit{Insts} \quad n.op = \textsc{Multiply} \quad N_{ck} = \{(n_c, k) \mid n_c.parm_k = n\}}{\mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_s\} \quad n_s.op \leftarrow \textsc{Rescale} \quad n_s.parm_1 \leftarrow n \quad n_s.parm_2 \leftarrow \min(\forall j,\ n.parm_j.scale) \quad \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_s}$$

$$\text{WATERLINE} - \text{RESCALE}\ \frac{n \in \mathit{Insts} \quad n.op = \textsc{Multiply} \quad N_{ck} = \{(n_c, k) \mid n_c.parm_k = n\} \quad (n.parm_1.scale * n.parm_2.scale)/s_f \geq \max(\forall n_j \in \{\mathit{Consts}, \mathit{Inputs}\},\ n_j.scale)}{\mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_s\} \quad n_s.op \leftarrow \textsc{Rescale} \quad n_s.parm_1 \leftarrow n \quad n_s.parm_2 \leftarrow s_f \quad \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_s}$$

$$\text{LAZY} - \text{MODSWITCH}\ \frac{n \in \mathit{Insts} \quad n.op \in \{\textsc{Add}, \textsc{Sub}, \textsc{Multiply}\} \quad n.parm_i.level > n.parm_j.level}{\mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_m\} \quad n_m.op \leftarrow \textsc{ModSwitch} \quad n_m.parm_1 \leftarrow n.parm_j \quad n.parm_j \leftarrow n_m}$$

$$\text{EAGER} - \text{MODSWITCH}\ \frac{n \in \{\mathit{Insts}, \mathit{Consts}, \mathit{Inputs}\} \quad n_c^1.parm_i = n \quad n_c^2.parm_j = n \quad n_c^1.parm_i.rlevel > n_c^2.parm_j.rlevel \quad N_{ck} = \{(n_c, k) \mid n_c.parm_k = n \wedge n_c.parm_k.rlevel = n_c^2.parm_j.rlevel\}}{\mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_m\} \quad n_m.op \leftarrow \textsc{ModSwitch} \quad n_m.parm_1 \leftarrow n \quad \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_m}$$

$$\text{MATCH} - \text{SCALE}\ \frac{n \in \mathit{Insts} \quad n.op \in \{\textsc{Add}, \textsc{Sub}\} \quad n.parm_i.scale > n.parm_j.scale}{\mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_t\} \quad \mathit{Consts} \leftarrow \mathit{Consts} \cup \{n_c\} \quad n_c.value \leftarrow n.parm_i.scale/n.parm_j.scale \quad n_t.op \leftarrow \textsc{Multiply} \quad n_t.parm_1 \leftarrow n.parm_j \quad n_t.parm_2 \leftarrow n_c \quad n.parm_j \leftarrow n_t}$$

$$\text{RELINEARIZE}\ \frac{n \in \mathit{Insts} \quad n.op = \textsc{Multiply} \quad n.parm_1.type = n.parm_2.type = \textbf{Cipher} \quad N_{ck} = \{(n_c, k) \mid n_c.parm_k = n\}}{\mathit{Insts} \leftarrow \mathit{Insts} \cup \{n_l\} \quad n_l.op \leftarrow \textsc{Relinearize} \quad n_l.parm_1 \leftarrow n \quad \forall (n_c, k) \in N_{ck},\ n_c.parm_k \leftarrow n_l}$$

Figure 3. Graph rewriting rules (each rule is a transformation pass) in EVA (s_f: maximum allowed rescale value).

Constrained-Optimization Problem: The only nodes that modify a ciphertext's coefficient modulus are Rescale and ModSwitch nodes; that is, they are the only ones whose output ciphertext has a different coefficient modulus than that of their input ciphertext(s). Therefore, the coefficient modulus of the output of a node depends only on the Rescale and ModSwitch nodes in the path from the root to that node. To illustrate their relation, we define the term *rescale chain*.

Definition 5.1. Given a directed acyclic graph G = (V, E):

For $n_1, n_2 \in V$, n_1 is a *parent* of n_2 if $\exists (n_1, n_2) \in E$.

A node $r \in V$ is a *root* if $r.type$ = **Cipher** and $\nexists n \in V$ s.t. n is a parent of r.

Definition 5.2. Given a directed acyclic graph G = (V, E):

A *path* p from a node $n_0 \in V$ to a node $n \in V$ is a sequence of nodes $p_0, ..., p_l$ s.t. $p_0 = n_0$, $p_l = n$, and $\forall 0 \leq i < l, p_i \in V$ and p_i is a parent of p_{i+1}. A path p is said to be *simple* if $\forall 0 < i < l, p_i.op \neq$ Rescale and $p_i.op \neq$ ModSwitch.

Definition 5.3. Given a directed acyclic graph G = (V, E):

A *rescale path* p to a node $n \in V$ is a sequence of nodes $p_0, ..., p_l$ s.t. ($\forall 0 \leq i \leq l, p_i.op \in$ {Rescale, ModSwitch}), $\exists$ a simple path from a root to p_0, $\exists$ a simple path from p_l to n, ($\forall 0 \leq i < l, \exists$ a simple path from p_i to p_{i+1}), and ($n.op$ = Rescale or $n.op$ = ModSwitch) $\implies$ $(p_l = n)$.

A *rescale chain* of a node $n \in V$ is a vector c s.t. $\exists$ a rescale path p to n and ($\forall 0 \leq i < |p|$, ($p_i.op$ = ModSwitch $\implies$ $c_i = \infty$) and ($p_i.op$ = Rescale $\implies$ $c_i = p_i.parm_2.value$)). Note that ∞ is used here to distinguish ModSwitch from Rescale in the rescale chain.

A rescale chain c of a node n and c' of a node n' are *equal* if ($|c| = |c'|$ and ($\forall 0 \leq i < |c|$, $c_i = c'_i$ or $c_i = \infty$ or $c'_i = \infty$)).

A rescale chain c of a node $n \in V$ is *conforming* if $\forall$ rescale chain c' of n, c is equal to c'.

Note that all the roots in the graph have the same coefficient modulus. Therefore, for nodes n_1 and n_2, the coefficient modulus of the output of n_1 is equal to that of n_2 if and only if there exists conforming rescale chains for n_1 and n_2, and the conforming rescale chain of n_1 is equal to that of n_2. Thus, we need to solve two problems simultaneously:

- Constraints: Ensure the conforming rescale chains of the parents of any Multiply or Add node are equal.
- Optimization: Minimize the length of the rescale chain of every node.

Outline: In general, the constraints problem can be solved in two steps:

- Insert Rescale in a pass (to reduce exponential growth of scale and noise).
- Insert ModSwitch in another pass so that the constraints are satisfied.

The challenge is in solving this problem in this way, while yielding the desired optimization.

Always Rescale Insertion: A naive approach of inserting Rescale is to insert it after every Multiply of **Cipher**

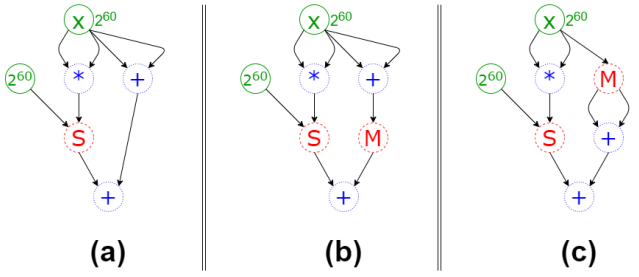

Figure 4. $x^2 + x + x$ in EVA: (a) after WATERLINE-RESCALE (b) after WATERLINE-RESCALE & LAZY-MODSWITCH; (c) after WATERLINE-RESCALE & EAGER-MODSWITCH.

nodes. We call this approach as *always rescale* and define it in the ALWAY-RESCALE rewrite rule in Figure 3. Consider the example in Figure 1(a). Applying this rewrite rule on this example yields the graph in Figure 1(b). For some MULTIPLY nodes (e.g., the bottom one), the conforming rescale chains of their parents do not exist or do not match. To satisfy these constraints, MODSWITCH nodes can be inserted appropriately, as shown in Figure 1(c) (we omit defining this rule because it would require multiple forward passes). The conforming rescale chain length for the output is now more more than the multiplicative depth of the graph. Thus, *always rescale* and its corresponding modswitch insertion may lead to a larger coefficient modulus (both in the number of elements and their product) than not inserting any of them.

Insight: Consider that all the roots in the graph have the same scale s. For example in Figure 1(a), let $x.scale = 2^{30}$ instead of 2^{60}. Then, after *always rescale* (replace 2^{60} with 2^{30} in Figure 1(b)), the only difference between the rescale chains of a node n would be their length and not the values in it. This is the case even when roots may have different scales as long as all RESCALE nodes rescale by the same value s. Then, a conforming rescale chain c_n for n can be obtained by adding MODSWITCH nodes in the smaller chain(s). Thus, $|c_n|$ would not be greater than the multiplicative depth of n. The first key insight of EVA is that using the same rescale value for all RESCALE nodes ensures that $|c_o|$ cannot be greater than the multiplicative depth of o (a tight upper bound). The multiplicative depth of a node n is not a tight lower bound for $|c_n|$, as shown in Figure 1(d). The second key insight of EVA is that using the maximum rescale value s_f (satisfying Constraint 4) for all RESCALE nodes minimizes $|c_o|$ because it minimizes the number of RESCALE nodes in any path.

Waterline Rescale Insertion: Based on our insights, the value to rescale is fixed to s_f (= 2^{60} in SEAL). That does not address the question of when to insert RESCALE nodes. If the scale after RESCALE becomes too small, then the computed message may lose accuracy irrevocably. We call the minimum required scale as *waterline* and use s_w to denote it. We choose s_w to be maximum of the scales of all roots. Consider a MULTIPLY node n whose scale after multiplication is s_n. Then, a RESCALE in inserted between n and its children only if the scale after RESCALE is above the *waterline*, i.e., $(s_n/s_f) \geq s_w$. We call this approach as *waterline rescale* and define the WATERLINE-RESCALE rewrite rule in Figure 3. This rule transforms the graph in Figure 1(a) to the one in Figure 1(d).

ModSwitch Insertion: For a node n, let $n.level$ denote its conforming rescale chain length. Let $n.rlevel$ denote the conforming rescale chain length of n in the transpose graph. A naive way to insert MODSWITCH is to find a ADD or MULTIPLY node for which *level* of the parents do not match and insert the appropriate number of MODSWITCH nodes between one of the parents and the node. We call this *lazy* insertion and define the LAZY-MODSWITCH rewrite rule in Figure 3. We call inserting it at the earliest feasible edge in the graph as *eager* insertion. The EAGER-MODSWITCH rewrite rule (Figure 3) finds a node for which *rlevel* of the children do not match and inserts the appropriate number of MODSWITCH nodes between some of the children and itself. If the *rlevel* of the roots do not match, then there is another rule (omitted in Figure 3 for brevity) that inserts the appropriate MODSWITCH nodes between some of the roots and their children.

Consider the $x^2 + x + x$ example in Figure 4(a). Applying the LAZY-MODSWITCH and EAGER-MODSWITCH rewrite rules yields the graphs in Figures 4(b) and (c) respectively. The operands of ADD after eager insertion use a smaller coefficient modulus than after lazy insertion, so ADD would be faster if eager insertion is used. Thus, eager insertion leads to similar or more efficient code than lazy insertion.

Matching Scales: As illustrated in Section 4.2, it is easy to match scales of parents of ADD by multiplying one of the parents and 1 with the appropriate scale. The MATCH-SCALE rewrite rule (Figure 3) takes this simple approach to satisfy Constraint 2 while avoiding introduction of any additional RESCALE or MODSWITCH. For the example graph in Figure 2(a), applying this rewrite rule transforms the graph into the one in Figure 2(c).

Optimality: EVA selects encryption parameters (see Section 6.2) s.t. $r = \max(\forall o \in \{Outputs\}, 1 + |c_o| + \lceil \frac{o.scale * s_o}{s_f} \rceil)$, where s_o is the desired scale for the output o. WATERLINE-RESCALE is the only pass that determines $|c_o|$ and $o.scale$ for any output o (neither LAZY-MODSWITCH nor MATCH-SCALE modify that). If $|c_o|$ is decreased by 1 (an element s_f from c_o is removed), then $o.scale$ would increase by at least s_f, so it would not decrease r. Due to *waterline rescale*, $o.scale < s_w * s_f$, so RESCALE cannot be inserted to reduce $o.scale$ by at least s_f (because the minimum required scale is s_w). Thus, EVA yields the minimal or optimal r.

6 Analysis in EVA Compiler

In this section, we briefly describe our graph traversal framework (Section 6.1) and a few analysis passes (Section 6.2).

6.1 Graph Traversal Framework and Executor

EVA's graph traversal framework allows either a forward traversal or a backward traversal of the graph. In the forward traversal pass, a node is visited only after all its parents are visited. Similarly, in the backward traversal pass, a node is visited only after all its children are visited. Graph traversals do not modify the structure of the graph (unlike graph rewriting) but a state on each node can be maintained.

Parallel Implementation: We implement an executor for the generated EVA program using the traversal framework. A node is said to be *ready* or *active* if all its parents (or children) in forward (or backward) pass have already been visited. These active nodes can be scheduled to execute in parallel as each active node only updates its own state (i.e., there are no conflicts). For example in Figure 1(e), the parents of the bottom Multiply can execute in parallel. Each FHE instruction (node) can take a significant amount of time to execute, so it is useful to exploit parallelism among FHE instructions. The EVA executor automatically parallelizes the generated EVA program by implementing a parallel graph traversal using the Galois [20, 37] parallel library.

A node is said to *retire* if all its children (or parents) in forward (or backward) pass have already been visited. The state for the retired nodes will no longer be accessed, so it can be reused for other nodes. In Figure 1(e), the ciphertext for x^2 can be reused after the Relinearize is executed. The EVA executor automatically reuses the memory used for encrypted messages, thereby reducing the memory consumed.

6.2 Analysis Passes

Validation Passes: We implement a validation pass for each of the constraints in Section 4.2. All are forward passes. The first pass computes the rescale chains for each node and asserts that it is conforming. It also asserts that the conforming rescale chains of parents of Add and Multiply match, satisfying Constraint 1. The second and third passes compute a *scale* and *num_polynomials* for each node respectively and assert that Constraint 2 and 3 is satisfied respectively. If any assertion fails, an exception in thrown at compile-time. Thus, these passes elide runtime exceptions thrown by SEAL.

Encryption Parameter Selection Pass: Akin to encryption selection in CHET [18], the encryption parameter selection pass in EVA computes the conforming *rescale chain* and the scale for each node. For each leaf or output o after the pass, let c_o be the conforming rescale chains of o without ∞ in it and let $s'_o = s_o * o.scale$, where s_o is the desired output scale. s'_o is factorized into $s_0 * s_1 * \ldots * s_k$ such that s_k is a power-of-two, $s_k \leq s_f$ ($= 2^{60}$ in SEAL), and $\forall 0 \leq i < k, s_i = s_f$. Let $|s'_o|$ denote the number of factors of s'_o. Then EVA finds the output m with the maximum $|c_m| + |s'_m|$. The factors of s_m are appended to c_m and s_f (the *special prime* that is consumed during encryption) in inserted at the beginning of c_m. For each element s in c_m, $\log_2 s$ is applied to obtain a vector of bit sizes, which is then returned.

```
from EVA import *
def sqrt(x):
  return x*constant(scale, 2.214) +
    (x**2)*constant(scale, -1.098) +
    (x**3)*constant(scale, 0.173)
program = Program(vec_size=64*64)
scale = 30
with program:
  image = inputEncrypted(scale)
  F = [[-1, 0, 1],
       [-2, 0, 2],
       [-1, 0, 1]]
  for i in range(3):
    for j in range(3):
      rot = image << (i*64+j)
      h = rot * constant(scale, F[i][j])
      v = rot * constant(scale, F[j][i])
      first = i == 0 and j == 0
      Ix = h if first else Ix + h
      Iy = v if first else Iy + v
  d = sqrt(Ix**2 + Iy**2)
  output(d, scale)
```

Figure 5. PyEVA program for Sobel filtering 64×64 images. The `sqrt` function evaluates a 3rd degree polynomial approximation of square root.

Rotation Keys Selection Pass: Similar to rotation keys selection in CHET [18], EVA's rotation keys selection pass computes and returns the set of unique step counts used among all RotateLeft and RotateRight nodes in the graph.

7 Frontends of EVA

The various transformations described so far for compiling an input EVA program into an executable EVA program make up the *backend* in the EVA compiler framework. In this section, we describe two *frontends* for EVA, that make it easy to write programs for EVA.

7.1 PyEVA

We have built a general-purpose frontend for EVA as a DSL embedded into Python, called PyEVA. Consider the PyEVA program in Figure 5 for Sobel filtering, which is a form of edge detection in image processing. The **`class`** `Program` is a wrapper for the Protocol Buffer [23] format for EVA programs mentioned in Section 3. It includes a context manager, such that inside a **`with`** `program:` block all operations are recorded in `program`. For example, the `inputEncrypted` function inserts an input node of type **Cipher** into the program currently in context and additionally returns an instance

of `class Expr`, which stores a reference to the input node. The expression additionally overrides Python operators to provide the simple syntax seen here.

7.2 EVA for Neural Network Inference

CHET [18] is a compiler for evaluating neural networks on encrypted inputs. The CHET compiler receives a neural network as a graph of high-level tensor operations, and through its kernel implementations, analyzes and executes these neural networks against FHE libraries. CHET lacks a proper backend and operates more as an interpreter coupled with automatically chosen high-level execution strategies.

We have obtained the CHET source code and modified it to use the EVA compiler as a backend. CHET uses an interface called *Homomorphic Instruction Set Architecture* (HISA) as a common abstraction for different FHE libraries. In order to make CHET generate EVA programs, we introduce a new HISA implementation that instead of calling homomorphic operations inserts instructions into an EVA program. This decouples the generation of the program from its execution. We make use of CHET's data layout selection optimization, but not its encryption parameter selection functionality, as this is already provided in EVA. Thus, EVA subsumes CHET.

8 Experimental Evaluation

In this section, we first describe our experimental setup (Section 8.1). We then describe our evaluation of homomorphic neural network inference (Section 8.2) and homomorphic arithmetic, statistical machine learning, and image processing applications (Section 8.3).

8.1 Experimental Setup

All experiments were conducted on a 4 socket machine with Intel Xeon Gold 5120 2.2GHz CPU with 56 cores (14 cores per socket) and 190GB memory. Our evaluation of all applications uses GCC 8.1 and SEAL v3.3.1 [41], which implements the RNS variant of the CKKS scheme [12]. All experiments use the default 128-bit security level.

We evaluate a simple arithmetic application to compute the path length in 3-dimensional space. We also evaluate applications in statistical machine learning, image processing, and deep neural network (DNN) inferencing using the frontends that we built on top of EVA (Section 7). For DNN inferencing, we compare EVA with the state-of-the-art compiler for homomorphic DNN inferencing, CHET [18], which has been shown to outperform hand-tuned codes. For the other applications, no suitable compiler exists for comparison. Hand-written codes also do no exist as it is very tedious to write them manually. We evaluate these applications using EVA to show that EVA yields good performance with little programming effort. For DNN inferencing, the accuracy reported is for all test inputs, whereas all the other results reported are an average over the first 20 test inputs. For the other applications, all results reported are an average over 20 different randomly generated inputs.

Table 3. Deep Neural Networks used in our evaluation.

Network	No. of layers			# FP	Accuracy(%)
	Conv	FC	Act	operations	
LeNet-5-small	2	2	4	159960	98.45
LeNet-5-medium	2	2	4	5791168	99.11
LeNet-5-large	2	2	4	21385674	99.30
Industrial	5	2	6	-	-
SqueezeNet-CIFAR	10	0	9	37759754	79.38

Table 4. Programmer-specified input and output scaling factors used for both CHET and EVA, and the accuracy of homomorphic inference in CHET and EVA (all test inputs).

Model	Input Scale ($\log P$)			Output	Accuracy(%)	
	Cipher	Vector	Scalar	Scale	CHET	EVA
LeNet-5-small	25	15	10	30	98.42	98.45
LeNet-5-medium	25	15	10	30	99.07	99.09
LeNet-5-large	25	20	10	25	99.34	99.32
Industrial	30	15	10	30	-	-
SqueezeNet-CIFAR	25	15	10	30	79.31	79.34

8.2 Deep Neural Network (DNN) Inference

Networks: We evaluate five deep neural network (DNN) architectures for image classification tasks that are summarized in Table 3:

- The three **LeNet-5** networks are all for the MNIST [32] dataset, which vary in the number of neurons. The largest one matches the one used in the TensorFlow's tutorials [42].
- **Industrial** is a network from an industry partner for privacy-sensitive binary classification of images.
- **SqueezeNet-CIFAR** is a network for the CIFAR-10 dataset [31] that uses 4 Fire-modules [15] and follows the SqueezeNet [27] architecture.

We obtain these networks (and the models) from the authors of CHET, so they match the networks evaluated in their paper [18]. Industrial is a FHE-compatible neural network that is proprietary, so the authors gave us only the network structure without the trained model (weights) or the test datasets. We evaluate this network using randomly generated numbers (between -1 and 1) for the model and the images. All the other networks were made FHE-compatible by CHET authors using average-pooling and polynomial activations instead of max-pooling and ReLU activations. Table 3 lists the accuracies we observed for these networks using unencrypted inference on the test datasets. We evaluate encrypted image inference with a batch size of 1 (latency).

Table 5. Average latency (s) of CHET and EVA on 56 threads.

Model	CHET	EVA	Speedup from EVA
LeNet-5-small	3.7	0.6	6.2
LeNet-5-medium	5.8	1.2	4.8
LeNet-5-large	23.3	5.6	4.2
Median	70.4	9.6	7.3
SqueezeNet-CIFAR	344.7	72.7	4.7

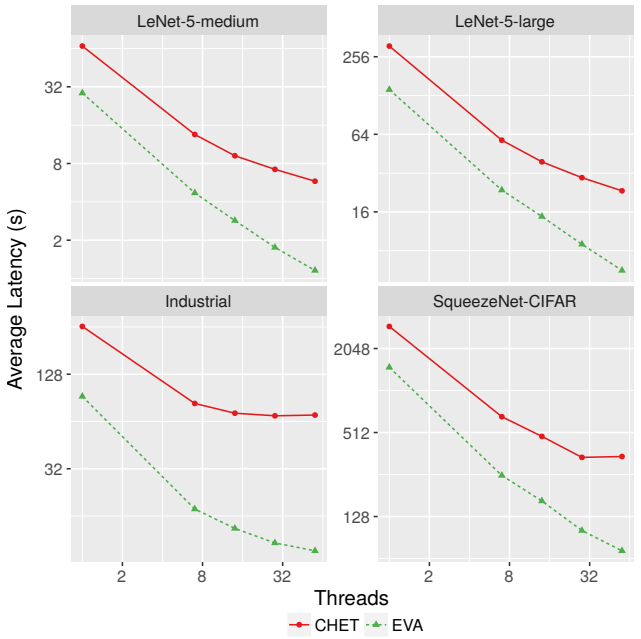

Figure 6. Strong scaling of CHET and EVA (log-log scale).

Scaling Factors: The scaling factors, or scales in short, must be chosen by the user. For each network (and model), we use CHET's profiling-guided optimization on the first 20 test images to choose the input scales as well as the desired output scale. There is only one output but there are many inputs. For the inputs, we choose one scale each for **Cipher**, **Vector**, and **Scalar** inputs. Both CHET and EVA use the same scales, as shown in Table 4. The scales impact both performance and accuracy. We evaluate CHET and EVA on all test images using these scales and report the accuracy achieved by fully-homomorphic inference in Table 4. There is negligible difference between their accuracy and the accuracy of unencrypted inference (Table 3). Higher values of scaling factors may improve the accuracy, but will also increase the latency of homomorphic inference.

Comparison with CHET Compiler: Table 5 shows that EVA is at least 4× faster than CHET on 56 threads for all networks. Note that the average latency of CHET is slower than that reported in their paper [18]. This could be due to differences in the experimental setup. The input and output scales they use are different, so is the SEAL version (3.1 vs. 3.3.1). We suspect the machine differences to be the primary reason for the slowdown because they use smaller number of heavier cores (16 3.2GHz cores vs. 56 2.2GHz cores). In any case, our comparison of CHET and EVA is fair because both use the same input and output scales, SEAL version, Channel-Height-Width (CHW) data layout, and hardware. Both CHET and EVA perform similar encryption parameters and rotation keys selection. The differences between CHET and EVA are solely due to the benefits that accrue from EVA's low-level optimizations.

Table 6. Encryption parameters selected by CHET and EVA (where $Q = \prod_{i=1}^{r} Q_i$).

Model	CHET			EVA		
	$\log_2 N$	$\log_2 Q$	r	$\log_2 N$	$\log_2 Q$	r
LeNet-5-small	15	480	8	14	360	6
LeNet-5-medium	15	480	8	14	360	6
LeNet-5-large	15	740	13	15	480	8
Industrial	16	1222	21	15	810	14
SqueezeNet-CIFAR	16	1740	29	16	1225	21

CHET relies on an expert-optimized library of homomorphic tensor kernels, where each kernel (1) includes FHE-specific instructions and (2) is explicitly parallelized. However, even experts cannot optimize or parallelize across different kernels as that information is not available to them. In contrast, EVA uses a library of vectorized tensor kernels and automatically (1) inserts FHE-specific instructions using global analysis and (2) parallelizes the execution of different instructions across kernels. Due to these optimizations, *EVA is on average* 5.3× *faster than CHET*. On a single thread (Figure 6), EVA is on average 2.3× faster than CHET and this is solely due to better placement of FHE-specific instructions. The rest of the improvement on 56 threads (2.3× on average) is due to better parallelization in EVA.

Both CHET and EVA have similar Relinearize placement. However, they differ in the placement of the other FHE-specific instructions — Rescale and ModSwitch. These instructions directly impact the encryption parameters (both CHET and EVA use a similar encryption parameter selection pass). We report the encryption parameters selected by CHET and EVA in Table 6. EVA selects much smaller coefficient modulus, both in terms of the number of elements r in it and their product Q. Consequently, the polynomial modulus N is one power-of-2 lower in all networks, except LeNet-5-large. Reducing N and r reduces the cost (and the memory) of each homomorphic operation (and ciphertext) significantly. In CHET, Rescale and ModSwitch used by the experts for a given tensor kernel may be sub-optimal for the program. On the other hand, EVA performs global (interprocedural) analysis to minimize the length of the coefficient modulus, yielding much smaller encryption parameters.

Table 7. Compilation, encryption context (context), encryption, and decryption time for EVA.

Model	Time (s)			
	Compilation	Context	Encrypt	Decrypt
LeNet-5-small	0.14	1.21	0.03	0.01
LeNet-5-medium	0.50	1.26	0.03	0.01
LeNet-5-large	1.13	7.24	0.08	0.02
Industrial	0.59	15.70	0.12	0.03
SqueezeNet-CIFAR	4.06	160.82	0.42	0.26

Table 8. Evaluation of EVA for fully-homomorphic arithmetic, statistical machine learning, and image processing applications on 1 thread (LoC: lines of code).

Application	Vector Size	LoC	Time (s)
3-dimensional Path Length	4096	45	0.394
Linear Regression	2048	10	0.027
Polynomial Regression	4096	15	0.104
Multivariate Regression	2048	15	0.094
Sobel Filter Detection	4096	35	0.511
Harris Corner Detection	4096	40	1.004

To understand the differences in parallelization, we evaluated CHET and EVA on 1, 7, 14, 28, and 56 threads. Figure 6 shows the strong scaling. We omit LeNet-5-small because it takes too little time, even on 1 thread. It is apparent that EVA scales much better than CHET. The parallelization in CHET is within a tensor operation or kernel using OpenMP. Such static, *bulk-synchronous* schedule limits the available parallelism. In contrast, EVA dynamically schedules the directed acyclic graph of EVA (or SEAL) operations asynchronously. Thus, it exploits the parallelism available across tensor kernels, resulting in much better scaling. The average speedup of EVA on 56 threads over EVA on 1 thread is 18.6× (excluding LeNet-5-small).

Comparison with Hand-Written LoLa: LoLa [7] implements hand-tuned homomorphic inference for neural networks, but the networks they implement are different than the ones we evaluated (and the ones in CHET). Nonetheless, they implement networks for the MNIST and CIFAR-10 datasets.

For the MNIST dataset, LoLa implements the highly-tuned CryptoNets [21] network (which is similar in size to LeNet-5-small). This implementation has an average latency of 2.2 seconds and has an accuracy of 98.95%. EVA takes only 1.2 seconds on a much larger network, LeNet-5-medium, with a better accuracy of 99.09%. For the CIFAR-10 dataset, LoLa implements a custom network which takes 730 seconds and has an accuracy of 74.1%. EVA takes only 72.7 seconds on a much larger network with a better accuracy of 79.34%.

LoLa uses SEAL 2.3 (which implements BFV [19]) which is less efficient than SEAL 3.3.1 (which implements RNS-CKKS [12]) but much more easier to use. EVA is faster because it exploits a more efficient FHE scheme which is much more difficult to manually write code for. Thus, *EVA outperforms even highly tuned expert-written implementations like LoLa with very little programming effort.*

Compilation Time: We present the compilation time, encryption context time, encryption time, and decryption time for all networks in Table 7. The encryption context time includes the time to generate the public key, the secret key, the rotation keys, and the relinearization keys. This can take a lot of time, especially for large N, like in SqueezeNet-CIFAR. Compilation time, encryption time, and decryption time are negligible for all networks.

8.3 Arithmetic, Statistical Machine Learning, and Image Processing

We implemented several applications using PyEVA. To illustrate a simple arithmetic application, we implemented an application that computes the length of a given encrypted 3-dimensional path. This computation can be used as a kernel in several applications like in secure fitness tracking on mobiles. For statistical machine learning, we implemented linear regression, polynomial regression, and multi-variate regression on encrypted vectors. For image processing, we implemented Sobel filter detection and Harris corner detection on encrypted images. All these implementations took very few lines of code (< 50), as shown in Table 8.

Table 8 shows the execution time of these applications on encrypted data using 1 thread. Sobel filter detection takes half a second and Harris corner detection takes only a second. The rest take negligible time. We believe *Harris corner detection is one of the most complex programs that have been evaluated using CKKS.* EVA enables writing advanced applications in various domains with little programming effort, while providing excellent performance.

9 Related Work

Libraries for FHE. SEAL [41] implements RNS variants of two FHE schemes: BFV [19] and CKKS [12, 13]. HElib [25] implements two FHE schemes: BGV [6] and CKKS. PALISADE [38] is a framework that provides a general API for multiple FHE schemes including BFV, BGV, and CKKS. For BFV and CKKS, PALISADE is similar to SEAL as it only implements lower-level FHE primitives. On the other hand, EVA language abstracts batching-compatible FHE schemes like BFV, BGV, and CKKS while hiding cryptographic details from the programmer. Although EVA compiler currently generates code targeting only CKKS implementation in SEAL, it can be adapted to target other batching-compatible FHE scheme implementations or FHE libraries.

General-Purpose Compilers for FHE. To reduce the burden of writing FHE programs, general-purpose compilers have been proposed that target different FHE libraries. These compilers share many of the same goals as ours. Some of these compilers support general-purpose languages like Julia (cf. [2]), C++ (cf. [14]), and R (cf. [3]), whereas ALCHEMY [16] is the only one that provides its own general-purpose language. Unlike EVA, none of these languages are amenable to be a target for domain-specific compilers like CHET [18] because these languages do not support rotations on fixed power-of-two sized vectors. Nonetheless, techniques in these compilers (such as ALCHEMY's static type safety and error rate analysis) are orthogonal to our contributions in this paper and can be incorporated in EVA.

All prior general-purpose compilers target (libraries implementing) either the BFV scheme [19] or the BGV scheme [6]. In contrast, EVA targets (libraries implementing) the recent CKKS scheme [12, 13], which is much more difficult to write or generate code for (compared to BFV or BGV). For example, ALCHEMY supports the BGV scheme and would require significant changes to capture the semantics (e.g., Rescale) of CKKS. ALCHEMY always inserts ModSwitch after every ciphertext-ciphertext multiplication (using local analysis), which is not optimal for BGV (or BFV) and would not be correct for CKKS. EVA is the first general-purpose compiler for CKKS and it uses a graph rewriting framework to insert Rescale and ModSwitch operations correctly (using global analysis) so that the modulus chain length is optimal. These compiler passes in EVA can be incorporated in other general-purpose compilers (to target CKKS).

Domain-Specific Compilers for FHE. Some prior compilers for DNN inferencing [4, 5, 18] target CKKS. CHET [18] is a compiler for tensor programs that automates the selection of *data layouts* for mapping tensors to vectors of vectors. The nGraph-HE [5] project introduced an extension to the Intel nGraph [17] deep learning compiler that allowed data scientists to make use of FHE with minimal code changes. The nGraph-HE compiler uses run-time optimization (e.g., detection of special plaintext values) and compile-time optimizations (e.g., use of ISA-level parallelism, graph-level optimizations). nGraph-HE2 [4] is an extension of nGraph-HE that uses a hybrid computational model – the server interacts with the client to perform non-HE compatible operations, which increases the communication overhead. Moreover, unlike CHET and EVA, neither nGraph-HE nor nGraph-HE2 automatically select encryption parameters.

To hide the complexities of FHE operations, all existing domain-specific compilers rely on a runtime of high-level kernels which can be optimized by experts. However, experts are limited to information within a single kernel (like convolution) to optimize insertion of FHE-specific operations and to parallelize execution. In contrast, EVA optimizes insertion of FHE-specific operations by using global analysis and parallelizes FHE operations across kernels transparently. Therefore, CHET, nGraph-HE, and nGraph-HE2 can target EVA instead of the FHE scheme directly to benefit from such optimizations and we demonstrated this for CHET.

Compilers for MPC. Multi-party computation (MPC) [22, 44] is another technique for privacy-preserving computation. The existing MPC compilers are mostly general-purpose [24] and even though it is possible to use them for deep learning applications, it is hard to program against a general-purpose interface. The EzPC compiler is a machine learning compiler that combines arithmetic sharing and garbled circuits and operates in a two-party setting [9]. EzPC uses ABY as a cryptographic backend [35].

Privacy-Preserving Deep Learning. CryptoNets, one of the first systems for neural network inference using FHE [21] and the consequent work on LoLa, a low-latency CryptoNets [7], show the ever more practical use of FHE for deep learning. CryptoNets and LoLa however use kernels for neural networks that directly translate the operations to the cryptographic primitives of the FHE schemes. There are also other algorithms and cryptosystems specifically for deep learning that rely on FHE (CryptoDL [26], [8], [29]), MPC (Chameleon [39], DeepSecure [40], SecureML [36]), oblivious protocols (MiniONN [33]), or on hybrid approaches (Gazelle [30], SecureNN [43].) None of these provide the flexibility and the optimizations of a compiler approach.

10 Conclusions

This paper introduces a new language and intermediate representation called Encrypted Vector Arithmetic (EVA) for general-purpose Fully-Homomorphic Encryption (FHE) computation. EVA includes a Python frontend that can be used to write advanced programs with little programming effort, and it hides all the cryptographic details from the programmer. EVA includes an optimizing compiler that generates correct, secure, and efficient code, targeting the state-of-the-art SEAL library. EVA is also designed for easy targeting of domain specific languages. The state-of-the-art neural network inference compiler CHET, when re-targeted onto EVA, outperforms its unmodified version by 5.3× on average. EVA provides a solid foundation for a richer variety of FHE applications as well as domain-specific compilers and auto-vectorizing compilers for computing on encrypted data.

Acknowledgments

This research was supported by the NSF grants 1406355, 1618425, 1705092, 1725322, and by the DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. We thank Keshav Pingali for his support. We thank the anonymous reviewers and in particular our shepherd, Petar Tsankov, for their many suggestions in improving this paper.

References

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org, Toronto, Canada.

[2] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (London, United Kingdom) *(WAHC'19)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/3338469.3358945

[3] Louis JM Aslett, Pedro M Esperança, and Chris C Holmes. 2015. A review of homomorphic encryption and software tools for encrypted statistical machine learning. *arXiv preprint arXiv:1508.06574* (2015).

[4] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*.

[5] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, Shafi Goldwasser (Ed.). Association for Computing Machinery, Cambridge, MA, USA, 309–325. https://doi.org/10.1145/2090236.2090262

[7] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. Low Latency Privacy Preserving Inference. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.).

[8] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. 2017. Privacy-Preserving Classification on Deep Neural Network. Cryptology ePrint Archive, Report 2017/035. http://eprint.iacr.org/2017/035.

[9] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *IEEE European Symposium on Security and Privacy, EuroS&P*.

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

[11] 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.

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A Full RNS Variant of Approximate Homomorphic Encryption. In *SAC 2018: 25th Annual International Workshop on Selected Areas in Cryptography (Lecture Notes in Computer Science)*, Carlos Cid and Michael J. Jacobson Jr: (Eds.), Vol. 11349. Springer, Heidelberg, Germany, Calgary, AB, Canada, 347–368. https://doi.org/10.1007/978-3-030-10970-7_16

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017, Part I (Lecture Notes in Computer Science)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.), Vol. 10624. Springer, Heidelberg, Germany, Hong Kong, China, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[14] Cingulata 2018. Cingulata. https://github.com/CEA-LIST/Cingulata.

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

[16] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1020–1037. https://doi.org/10.1145/3243734.3243828

[17] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *CoRR* abs/1801.08058 (2018). arXiv:1801.08058 http://arxiv.org/abs/1801.08058

[18] 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*.

[19] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.

[20] Galois System 2019. Galois System. http://iss.oden.utexas.edu/?p=projects/galois

[21] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of The 33rd International Conference on Machine Learning, ICML*.

[22] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th Annual ACM Symposium on Theory of Computing*, Alfred Aho (Ed.). ACM Press, New York City, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[23] Google Inc. [n.d.]. Protocol Buffer. https://developers.google.com/protocol-buffers. Google Inc.

[24] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1220–1237. https://doi.org/10.1109/SP.2019.00028

[25] HElib 2020. HElib. https://github.com/homenc/HElib.

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

[27] 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.

[28] Cryptography Lab in Seoul National University. [n.d.]. *Homomorphic Encryption for Arithmetic of Approximate Numbers (HEAAN)*.

[29] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. 2018. Secure Outsourced Matrix Computation and Application to Neural Networks. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 1209–1222. https://doi.org/10.1145/3243734.3243837

[30] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security 2018: 27th USENIX Security Symposium*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, Baltimore, MD, USA, 1651–1669.

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

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

[33] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 619–631. https://doi.org/10.1145/3133956.3134056

[34] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010 (Lecture Notes in Computer Science)*, Henri Gilbert (Ed.), Vol. 6110. Springer, Heidelberg, Germany, French Riviera, 1–23. https://doi.org/10.1007/978-3-642-13190-5_1

[35] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 35–52. https://doi.org/10.1145/3243734.3243760

[36] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 19–38. https://doi.org/10.1109/SP.2017.12

[37] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[38] PALISADE 2020. PALISADE Homomorphic Encryption Software Library. https://palisade-crypto.org/.

[39] 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 *ASIACCS 18: 13th ACM Symposium on Information, Computer and Communications Security*, Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim (Eds.). ACM Press, Incheon, Republic of Korea, 707–721.

[40] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. 2018. Deepsecure: Scalable Provably-secure Deep Learning. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3195970.3196023

[41] SEAL 2019. Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

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

[43] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (July 2019), 26–49. https://doi.org/10.2478/popets-2019-0035

[44] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Toronto, Ontario, Canada, 162–167. https://doi.org/10.1109/SFCS.1986.25