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

Roshan Dathathri[*], Blagovesta Kostova[†], Olli Saarikivi[‡], Wei Dai[‡], Kim Laine[‡], and Madanlal Musuvathi[‡]

[*]University of Texas at Austin, USA
roshan@cs.utexas.edu
[†]EPFL, Switzerland
blagovesta.pirelli@epfl.ch
[‡]Microsoft Research, USA
{olsaarik,wei.dai,kilai,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 implementation 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 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 $5.3\times$ *faster than those generated by CHET. We believe EVA would enable a wider adoption of FHE by making it easier to develop FHE applications and domain-specific FHE compilers.*

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 [13, 12] along with improved implementations have pushed FHE into the realm of practicality. For instance, when optimized appropriately, 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 [38, 26], 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 various 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 runtime errors that are common when programming FHE libraries manually.

EVA implements FHE-specific optimizations, such as optimally inserting operations like rescaling and modulus switching. EVA automatically reuses the memory used for encrypted messages, thereby reducing the memory consumed. We have built a compiler incorporating all these optimizations to generate efficient programs that run using the Microsoft SEAL [38] FHE library, which implements the RNS-variant of the CKKS scheme. We have also built an EVA executor that transparently parallelizes the generated program efficiently, allowing programs to scale well.

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

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 runtime of operations on higher-level constructs like tensors and images. In particular, our DNN compiler subsumes the recently proposed domain-specific compiler called CHET [17]. 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 the background on fully-homomorphic encryption. 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 Section 9 and 10.

2. Background and Motivation

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

2.1. Fully-Homomorphic Encryption

An FHE scheme includes four stages, key generation, encryption, evaluation, and decryption. Most of the efficient FHE schemes, for example, BGV [6], BFV [18], and CKKS [13], are constructed on the Ring Learning with Errors (RLWE) problem [32]. At the time of key generation, a polynomial ring of degree N with coefficients integers 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 or a vector of messages 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. Encryption also adds to a ciphertext a small random error that is later removed in decryption.

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.

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 evaluation circuit. When the errors get too large, ciphertexts become corrupted and cannot be decrypted, even with the correct secret key. Thus, to support efficient homomorphic evaluation of a circuit, one must optimize the circuit for a lower depth. Furthermore, the multiplicative depth of the circuit also determines how large N and Q must be, to ensure correct decryption while staying secure.

Relinearization: After each multiplication of ciphertexts, the resulting ciphertext consists of three polynomials, while freshly encrypted ciphertexts consist of only two polynomials. To prevent ciphertext sizes from growing indefinitely, an operation called relinearization is performed to reduce the number of polynomials in a ciphertext back to two. Relinearization is costly and their optimal placement is an NP-hard problem [10].

CKKS and Approximate Fixed-Point: The CKKS 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, which coupled with high enough scaling factors allows these errors to be hidden.

CKKS further features an operation called *rescaling* that scales down the fixed-point representation of a ciphertext. Consider a ciphertext that contains the encoding of 0.25 multiplied by the scale 2^{10} (a relatively low scale in CKKS). Its second power encode 0.0625 multiplied by the scale 2^{20}. Further powers would rapidly overflow modest values of the modulo 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.*

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 [38] 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.

Table 1: Types of values

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

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. Input programs use a subset of the language that omits details specific to homomorphic encryption, such as when to rescale. In this section, we describe this input format 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. The concatenation of two **Vector** values x and y is written $x + y$. 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. 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 opcodes that are inserted by the compiler.

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 nodes as outputs of the program (i.e., $Outputs \cap (Insts \cup Consts \cup Inputs) = Outputs$).

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 $I : Inputs \rightarrow$ **Vector** let $\mathscr{E}_{id}(n)$ be the function that recursively computes the value for node n using plaintext semantics and using $n.value$ and $I(n)$ for constants and inputs, respectively. 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:

$$\begin{aligned} P_{id} : \times_{n \in Inputs} n.type &\rightarrow \times_{n \in Outputs} \mathbf{Vector} \\ P_{id}(a^1, \ldots, a^{|Inputs|}) &= (\mathscr{E}_{id}(Outputs^1), \ldots, \mathscr{E}_{id}(Outputs^n)) \\ &\text{where } I(n) \equiv a^i \text{ s.t. } Inputs^i = n \\ &\text{and } n = |Outputs| \end{aligned}$$

These execution semantics hold for any encryption scheme, except that output is also encrypted.

The EVA language has a serialized format defined using Protocol Buffers [22], a language and platform neutral data serialization format. Additionally, the EVA language has an in-memory representation that is designed for efficient analysis and transformation, which is discussed in Section 4.

4. Overview of EVA Compiler

In this section, we briefly describe how to use the EVA compiler (Section 4.1). We then describe the constraints on the code generated by the EVA compiler (Section 4.2). Finally, we give an overview of the execution flow of the compiler (Section 4.3).

4.1. Using the Compiler

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. Note that we can make these instructions optional in the input and the compiler can handle it if they are present, but for the sake of exposition, we assume that the input does not have these instructions.

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 [17] can be used to select the appropriate values.

4.2. Motivation and Constraints

EVA compiler can be generalized to support any *batched* FHE scheme. Nevertheless, in the rest of this paper, we present EVA compiler specifically for the RNS variant of the CKKS scheme [12]. We use the SEAL [38] implementation of this scheme as an example throughout the paper. Targeting EVA for the CKKS scheme [13] or the HEAAN library [26] would be straightforward.

There is a one-to-one mapping between instructions in the EVA language (Table 2) and instructions in the RNS-CKKS scheme, save SUM (more on that later). 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 RNS-CKKS has a coefficient modulus (vector of primes) and a fixed-point scale associated with it. The following constraints apply for the binary instructions

Table 2: Instruction opcodes and their semantics

Opcode	Signature	Description	Format restrictions
Negate	**Cipher** → **Cipher**	Negate each element of the argument.	
Add	**Cipher** × **Cipher** → **Cipher**	Add arguments elementwise.	
Sub	**Cipher** × **Cipher** → **Cipher**	Subtract right argument from left argument elementwise.	
Multiply	**Cipher** × **Cipher** → **Cipher**	Multiply arguments elementwise	
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 (see Section 2).	Not in input
ModSwitch	**Cipher** → **Cipher**	Switch to the next modulus in the modulus chain (see Section 2).	Not in input
Rescale	**Cipher** × **Scalar** → **Cipher**	Rescale the ciphertext with the given divisor (see Section 2).	Not in input

involving two ciphertexts in the RNS-CKKS scheme:

$$\begin{gathered} n.parm_1.modulus = n.parm_2.modulus \\ \text{if } n.op \in \{\textsc{Add}, \textsc{Sub}, \textsc{Multiply}\} \\ n.parm_1.scale = n.parm_2.scale \\ \text{if } n.op \in \{\textsc{Add}, \textsc{Sub}\} \end{gathered} \tag{1}$$

Equation 1 shows the constraints on the input ciphertexts for certain instructions in the RNS-CKKS scheme; the other instructions do not have any constraints. In the rest of this paper, whenever we mention Add regarding constraints, it includes both Add and Sub.

We will use the following example in the rest of this section to illustrate the complications that arise due to the constraints on instructions. Consider a ciphertext x and computation $x^2 + x$. Let x have a scale s_x and the desired output scale be s_d. After Multiply, x^2 gets a scale of s_x^2. Consequently, Add is now trying to add ciphertexts with different scales, which would yield an exception.

One way to enforce that scales of the two operands of Add match is to multiply the operand with the lower scale and a constant 1 with the appropriate scale so that the product of the two scales yields the higher scale. In the example, if the computation is transformed to $x^2 + x * 1$ such that 1 has the scale s_x, then both operands of Add will have the same scale. Even though this is a feasible strategy, this may be insufficient for generating efficient code.

Without the use of Rescale instructions, the scales and the *noise* of the ciphertexts would grow exponentially with the multiplicative depth of the program and consequently, the product of the coefficient modulus required for the input would grow exponentially. Instead, using Rescale instructions ensures that they would only grow linearly with the multiplicative depth of the program. In the example, the output of $x^2 + x * 1$ has a scale of s_x^2. This requires the coefficient modulus of x to be at least $\{s_x^2, s_d\}$[1], where the last s_d is for the desired output scale. Instead, if the output of x^2 is rescaled, then x^2 would have a scale of s_x and can be added to x directly (without adding another multiplication). Thus, the output of $rescale(x^2) + x$ has a scale of s_x. This requires the coefficient modulus of x to be at least $\{s_x, s_x, s_d\}$; the first s_x would be *consumed* by the rescale.

Insertion of Rescale instructions may lead to violating the constraints of other instructions. In the transformed program $rescale(x^2) + x$ with $\{s_x, s_x, s_d\}$ as the coefficient modulus of x, the two operands of Add have coefficient modulus of $\{s_x, s_d\}$ amd $\{s_x, s_x, s_d\}$ because Rescale would have consumed s_x in the first operand. This violates the constraint that the coefficient modulus of Add operands must match. To resolve this, we can insert ModSwitch before the second operand, which just consumes s_x without changing the scale (unlike Rescale). Thus, the transformed program $rescale(x^2) + modswitch(x)$ would be correct and efficient.

For the sake of exposition, we omitted a few constraints in Equation 1. Firstly, another constraint on Multiply is that all ciphertext operands of Multiply must only have 2 polynomials. A Multiply of 2 ciphertexts results in 3 polynomials. Relinearize of this ciphertext yields a ciphertext with 2 polynomials. This must be done before the next Multiply. Secondly, the scalar operand for a Rescale must be $\leq 2^{60}$.

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.*

[1] In SEAL, if the coefficient modulus is $\{q_1, q_2, ..., q_l\}$, 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.

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($P_o$, $S_i$, $S_d$)
$R_s$ = DetermineRotations($P_i$, $S_i$)
```

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 as added 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 transformation program is validated to ensure that no constraints are violated. If any constraints are violated, then the compiler throws an exception. By doing this, the compiler ensures that executing the output program will never lead to a runtime exception in the FHE library. Finally, for 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 three graph transformation passes (Sections 5.2 and 5.3).

5.1. Graph Rewriting Framework

A graph transformation can be captured succinctly using graph *rewriting rules* (or term rewriting rules). These rules specify the transformation of a subgraph (or an expression) and the graph transformation consists of transforming all applicable subgraphs (or expressions) in the graph (or program). In other words, the rewriting rules specify local operations on a graph, and the graph transformation itself is composed of applying these local operations wherever needed. The order in which these local operations are applied may impact the correctness or efficiency of the transformation.

The nodes in the graph 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 rewriting rules may read and update the state. Moreover, the rewriting rules may be conditional on the state and properties of the nodes in the subgraph. Depending on the conditions, the rewriting rules may require (or prefer) to be applied in a particular order. Consider two orders: (1) forward pass from roots to leaves of the graph, or (2) backward pass from leaves to roots of the graph. 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 rewriting rules until quiescence (no change), but a single forward or backward pass might suffice.

EVA includes a graph rewriting framework for arbitrary rewriting rules for a subgraph that consists of a node along with its parents or children. Thus, EVA supports rewriting rules that specify how to transform a node and its neighbors. EVA supports both forward and backward passes. In the forward pass, a node is scheduled for rewriting only after all its parents have already been rewritten (note that the rewriting operation may not do any modifications if its condition does not hold). Similarly in backward pass, a node is scheduled for rewriting only after all its children have already been rewritten. In all graph transformations in EVA, a single forward or backward pass is sufficient.

In summary, a graph transformation consists of (1) the state on each node, (2) whether it is a forward pass or a backward pass, and (3) the graph rewriting rules. The graph rewriting rules consist of (1) the conditions on a subgraph consisting of a node and its neighbors, (2) the updates to the state, and (3) the transformation of the subgraph.

5.2. Relinearize Insertion Pass

Each ciphertext is represented as 2 or more polynomials. When two ciphertexts each with 2 polynomials are multiplied, it yields a ciphertext with 3 polynomials. SEAL does not support multiplication of a 3 polynomials ciphertext with another ciphertext, plaintext, or scalar. The RELINEARIZE instruction reduces a ciphertext from 3 polynomials to 2 polynomials. Thus, EVA must insert this instruction after MULTIPLY of two **Cipher** nodes and before another MULTIPLY.

The relinearization insertion pass requires no state on any node and can be implemented using either a forward pass or a backward pass. The rewriting rule is applied for a node n only if it is a MULTIPLY operation and if both its parents (or operands) have **Cipher** type. The transformation in the rule inserts a RELINEARIZE node n_r between the node n and its children. In other words, the new children of n will be only n_r

and the children of n_r will be the old children of n.

This pass *eagerly* inserts RELINEARIZE instructions soon after the appropriate MULTIPLY. There is a variant of the pass that *lazily* inserts RELINEARIZE instruction before the appropriate MULTIPLY. We implemented this variant, but omit its description for simplicity.

5.3. Rescale and ModSwitch Insertion Passes

Goal: The RESCALE and MODSWITCH nodes (or instructions) must be inserted such that they satisfy the constraints in Equation 1. There are two problems: (1) scales of parents (or operands) of ADD must match, and (2) coefficient moduli of parents of ADD and MULTIPLY must match. As described in Section 4.2, it is easy to resolve the first issue by adding a multiplication of one of the parents with the appropriate scale. We take this simple approach for matching scales and omit the details due to lack of space. The main problem is in resolving the second issue. 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 the constraints is sufficient for correctness, different choices lead to different coefficient modulus $\{Q_1, Q_2, ..., Q_r\}$, 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 get the optimal* r, which may or may not yield the optimal N.

Constrained-Optimization Problem: The only nodes that modify the 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 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 $\nexists n \in V$ such that n is a parent of r.*

Definition 2 *Given a directed acyclic graph G = (V, E):*

A path *p to a node $n \in V$ is a sequence of nodes $p_0, p_1, ..., p_l$ such that p_0 is a root, $p_l = n$, and $\forall 0 \le i < l, p_i \in V$ and p_i is a parent of p_{i+1}.*

A path p to a node $n \in V$ is said to be simple *if $\forall 0 < i < l, p_i.op \neq$ RESCALE and $p_i.op \neq$ MODSWITCH.*

Definition 3 *Given a directed acyclic graph G = (V, E):*

A rescale path *p to a node $n \in V$ is a sequence of nodes $p_0, p_1, ..., p_l$ such that ($\forall 0 \le i \le l, p_i.op =$ RESCALE or $p_i.op =$ MODSWITCH), $\exists$ a simple path from a root to p_0, $\exists$ a simple path from p_l to n, ($\forall 0 \le i < l, \exists$ a simple path from p_i to p_{i+1}), and ($p_l \neq n$) $\implies$ ($n.op \neq$ RESCALE and $n.op \neq$ MODSWITCH).*

A rescale chain *of a node $n \in V$ is a vector v such that $\exists$ a rescale path p and (($p_i.op =$ MODSWITCH) $\implies v_i = \infty$) and (($p_i.op =$ RESCALE) $\implies v_i = p_i.parm_2.value$). Note that ∞ is used here to distinguish* MODSWITCH *from* RESCALE *in the rescale chain.*

A rescale chain v of a node $n \in V$ is conforming *if $\forall$ rescale chain v' of n, $v_i = v'_i$ or $v_i = \infty$ or $v'_i = \infty$.*

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, RESCALE and MODSWITCH insertion passes aim 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. We call this approach as *always rescale*. This may lead to a larger coefficient modulus (both in the number of elements and their product). For example, consider ciphertexts x and y with a scale of $s_x = 2^{30}$ and $s_y = 2^{20}$ respectively, and say the computation is $x^2 + y^2$. If RESCALE is inserted after each multiplication, then x^2 and y^2 have a scale of s_x and s_y respectively. To ADD them, the scales must match and this can be resolved easily (like in Section 4.2). In addition, their coefficient modulus must match, and consequently, their rescale chains must match. This can be achieved by inserting MODSWITCH nodes appropriately (before y and after x^2) so that the rescale chains for both are $\{s_x, s_y\}$. For deeper circuits, this would require multiple passes and lead to much longer conforming rescale chains than the multiplicative depth of the graph (i.e., maximum number of MULTIPLY nodes in any path).

Insight: Consider that all the roots in the graph have the same scale s. Then, in the *always rescale* approach, the only difference between the rescale chains of a node n would be their length and not the values in it. A conforming rescale chain for n can be obtained by adding ModSwitch nodes in the smaller chain(s). Thus, the length of the conforming rescale chain of a node n would not be greater than the multiplicative depth of n. This is possible because all Rescale nodes rescale by the same value s. The first key insight of EVA is that it is sufficient to use the same rescale value for all Rescale nodes to give a tight bound on the length of the conforming rescale chain.

The multiplicative depth of a node is not necessarily the minimum length of its conforming rescale chain. For example, for the computation $(x^2)^2$, one can rescale soon after x^2 or after $(x^2)^2$ depending on the scale of x and the allowable rescale values in the FHE schemes. The second key insight of EVA is that the length of the conforming rescale chain is optimal (or minimal) if the largest allowed rescale value (which is $s = 2^{60}$ in SEAL) is used in all Rescale nodes.

Waterline Rescale Insertion: Based on our insights, the value to rescale is fixed to the maximum allowed value, which is denoted by s_f. That does not address the question of when to insert Rescale nodes. It is correct to insert a Rescale node only if the resulting scale (after rescale) is above a threshold or *waterline*. As different roots could have different scales, we choose the waterline to be maximum of all their values and denote this by s_w. Consider a Multiply node n whose scale after multiplication is s_n. Then, a Rescale in inserted between n and its children only if $(s_n/s_f) \geq s_w$. We call this approach as *waterline rescale*. It is optimal in minimizing r.

The Rescale insertion transformation is a forward pass that maintains a scale s_n on every node n. The scale s is updated as it would be during encrypted execution; Multiply multiplies the scale, Rescale divides the scale, and the rest copy the scale from their **Cipher** parents. The pass includes the above rewriting rule. It also includes another rewriting rule for Add to insert a Multiply after one of its operands only if their scales do not match.

This pass *eagerly* inserts Rescale instructions soon after the appropriate Multiply. There is a variant of the pass that *lazily* inserts Rescale instruction before the appropriate Multiply. We implemented this variant, but omit its description for simplicity.

ModSwitch Insertion: After the *waterline rescale* insertion pass, a naive way to insert ModSwitch is to determine whether the conforming rescale chains of the parents of a Add or a Multiply node match and if they do not, then insert the appropriate number of ModSwitch nodes between one of the parents and the node. This is sufficient to enforce the constraints. However, as it inserts ModSwitch just before it is needed, the parents might be using a higher coefficient modulus than required, leading to inefficient computation. Thus, we call this *lazy* insertion.

If ModSwitch is inserted earlier in the graph, then all its children will be faster due to a smaller coefficient modulus. We call inserting it at the earliest feasible edge in the graph as *eager* insertion. This graph transformation is a backward pass. It maintains a *reverse-level* l on each node n, which denotes the number of Rescale or ModSwitch nodes in all paths from n to leaves in the graph; Rescale and ModSwitch nodes increment l, while the rest copy l from their children. If children's l do not match, then appropriate ModSwitch nodes are inserted between n and the applicable children so that l of all children of n match. This ensures that all rescale chains in the transpose of the graph are conforming, which implies that all rescale chains in the graph are conforming.

6. Analysis in EVA Compiler

In this section, we describe a general graph traversal framework (Section 6.1) and briefly describe a few analysis passes (Section 6.2). The graph traversal framework is also used to implement an executor for the generated EVA program but we omit its description due to lack of space.

6.1. Graph Traversal Framework

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. Nonetheless, a state on each node can be maintained during the traversal. A single pass is sufficient to perform forward or backward data-flow analysis of the graph because the graph is acyclic. Execution of the graph is a forward traversal of the graph, so uses the same framework. A graph traversal pass can be succinctly captured by: (1) the state on each node (and its initial state), (2) whether it is a forward or backward pass, and (3) the state update for each EVA node (or instruction) based on its parents (or children) in forward (or backward) pass (similar to data-flow equations).

Parallel Implementation: 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). We implement such a parallel graph traversal in EVA using the Galois [35, 19] parallel library.

6.2. Analysis Passes

Validation Passes: We implement two passes to validate that the constraints in Section 4.2 are satisfied. Both are forward passes. In the first pass, we maintain a scale on each node; Multiply multiplies the scale, Rescale divides the scale, and the rest copy the scale from their **Cipher** parents. The pass asserts that both the parents of any Add node have the same scale. In the second pass, we maintain the *rescale* *chain* (vector of values) on each node; RESCALE node add its rescale value to, MODSWITCH node adds ∞ to, and the rest copy from the rescale chain of their parents. The pass asserts that both the rescale chains of parents of any ADD or MULTIPLY node are *conforming*. If the assertions failed, the pass fails and an exception in thrown at compile-time. The validation passes thus elide runtime exceptions in SEAL.

Encryption Parameter Selection Pass: Similar to encryption selection in CHET [17], the encryption parameter selection pass in EVA maintains the conforming *rescale chain* and the scale on each node. After the traversal, for each leaf, the leaf's scale and desired output scale are multiplied and the maximum one is chosen among them, denoted as s_m. Among the conforming rescale chains of the leaves in the graph, the maximum length one is chosen (without ∞ in the chain). To the maximum rescale chain, the maximum allowed rescale value (2^{60} in SEAL) in inserted at the beginning of the chain (it is called the *special prime*) because it is consumed during encryption. More rescale values are appended to this chain until s_m is consumed. In other words, s_m is factorized into $s_0 * s_1 * \ldots * s_k$ such that k is minimized and $\forall 0 \leq i \leq k, s_i \leq 2^{60}$ and s_i is a power-of-two. For each element s in the appended chain, $\log_2 s$ is applied to obtain a vector of bit sizes, which is then returned.

Rotation Keys Selection Pass: Similar to rotation keys selection in CHET [17], the rotation keys selection pass in EVA maintains a set of rotation steps on each node; ROTATELEFT and ROTATERIGHT insert their step count to (ROTATERIGHT step count is normalized to ROTATELEFT step count) and the rest copy their step count from the set of their parents. After the traversal, the union of the sets for all the leaves in the graph is returned.

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 1 for Sobel filtering, which is a form of edge detection in image processing. The **class** `Program` is a wrapper for the Protocol Buffer [22] 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.

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

7.2. EVA for Neural Network Inference

CHET [17] 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 SEAL v3.3.1 [38], that implements the RNS variant of the CKKS scheme [12]. All experiments use the default 128-bit security level. All results reported are an average over 20 different test inputs, unless otherwise specified.

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 [17], 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, like the homomorphic image processing applications. We evaluate these applications using EVA to show that EVA yields good performance with little programming effort.

8.2. Deep Neural Network (DNN) Inference

Networks: We evaluate a set of 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 [30] dataset, which vary in the number of neurons. The largest one matches the one used in the TensorFlow's tutorials [39].
- **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 [29] that uses 4 Fire-modules [15] and follows the SqueezeNet [25] architecture.

Table 3: Deep Neural Networks used in our evaluation.

Network	No. of layers			# FP	Accu-
	Conv	FC	Act	operations	racy(%)
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 achieved by fully-homomorphic inference in EVA (for all test datasets).

Model	Input Scale ($\log P$)			Output	Accu-
	Cipher	Vector	Scalar	Scale	racy(%)
LeNet-5-small	25	15	10	30	98.45
LeNet-5-medium	25	15	10	30	99.09
LeNet-5-large	25	20	10	25	99.29
Industrial	30	15	10	30	-
SqueezeNet-CIFAR	25	15	10	30	78.88

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

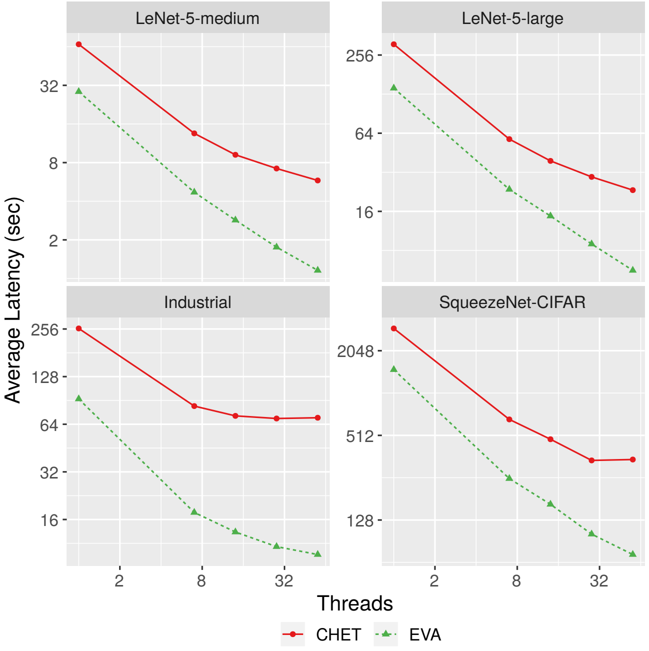

Figure 2: Strong scaling of CHET and EVA (log-log scale).

We obtain these networks (and the models) from the authors of CHET, so they match the networks evaluated in their paper [17]. 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).

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 EVA on all test images using these scales and report the accuracy of homomorphic inference in Table 4 (we do not evaluate CHET on all test images because it is much slower than EVA). When compared with the accuracy of unencrypted inference (Table 3), there is a negligible degradation. 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 [17]. 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. The differences between CHET and EVA are solely due to the benefits that accrue from EVA's low-level optimizations. Thus, *EVA is on average* 5.3× *faster than CHET.*

Strong Scaling: To understand the performance differences between CHET and EVA, we evaluated them on 1, 7, 14, 28, and 56 threads. Figure 2 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).

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

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
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

Encryption Parameters: EVA is much faster than CHET, even on 1 thread (by 2.3× on average). To understand this, 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 in it and their product. Consequently, the polynomial modulus is one power-of-2 lower in all networks, except LeNet-5-large. This reduction reduces the cost (and the memory) of each homomorphic operation (and ciphertext) significantly. CHET relies on an expert-optimized library of homomorphic tensor kernels. However, even experts cannot optimize across different kernels as that information is not available to them. Consequently, RESCALE and MODSWITCH used by these experts for a given tensor kernel may be sub-optimal for the program. On the other hand, EVA performs global (inter-procedural) analysis to minimize the length of the coefficient modulus, yielding much smaller encryption parameters.

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 [20] 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 that 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 78.88%.

LoLa uses SEAL 2.3 (which implements BFV [18]) 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 has been homomorphically evaluated*. EVA enables writing advanced applications in various domains with little programming effort, while providing excellent performance.

9. Related Work

Compilers for FHE: To reduce the burden of writing FHE programs, compilers have been proposed that target different FHE libraries. Some of these compilers support general purpose languages like Julia (cf. [2]), C++ (cf. [14]) and R (cf. [3]), but they are not amenable to incorporating domain-specific or target-specific optimizations, like EVA. None of these compilers target the recent CKKS scheme [13, 12] (or SEAL library [38]) which is more complex to write or generate code for.

Some of the existing domain-specific compilers [17, 5, 4] target CKKS, but they rely on expert-optimized runtime of high-level operations that hides the complexities of FHE operations. CHET [17] is a compiler for tensor programs that automates the selection of *data layouts*, and as we show, this can be used by a frontend of EVA. The nGraph-HE [5] project introduced an extension to the Intel nGraph [16] 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) to achieve a good performance. 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, neither nGraph-HE nor nGraph-HE2 introduce automatic encryption parameter selection, like EVA. In any case, CHET, nGraph-HE, and nGraph-HE2 can target EVA instead of the FHE scheme directly to benefit from low-level optimizations.

No existing compiler automatically inserts RELINEARIZE, RESCALE, or MODSWITCH operations. EVA not only inserts them but also minimizes the coefficient modulus chain length.

Compilers for MPC: Multi-party computation (MPC) [21, 41] is another technique for privacy-preserving computation. The existing MPC compilers are mostly general-purpose [23] 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 [33].

Privacy-Preserving Deep Learning: CryptoNets, one of the first systems for neural network inference using FHE [20] 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 [24], [8], [27]), MPC (Chameleon [36], DeepSecure [37], SecureML [34]), oblivious protocols (MiniONN [31]), or on hybrid approaches (Gazelle [28], SecureNN [40].) 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 and domain-specific FHE compilers.

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. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[2] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing &#38; Applied Homomorphic Cryptography*, WAHC'19, pages 57–68, New York, NY, USA, 2019. ACM.

[3] Louis JM Aslett, Pedro M Esperança, and Chris C Holmes. 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. nGraph-HE2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[5] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE: A graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

[7] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML*, 2019.

[8] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint Archive, Report 2017/035, 2017. `http://eprint.iacr.org/2017/035`.

[9] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *IEEE European Symposium on Security and Privacy, EuroS&P*, 2019.

[10] Hao Chen. 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. A full RNS variant of approximate homomorphic encryption. In *Selected Areas in Cryptography – SAC 2018*. Springer, 2018. LNCS 11349.

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr:, editors, *SAC 2018: 25th Annual International Workshop on Selected Areas in Cryptography*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368, Calgary, AB, Canada, August 15–17, 2019. Springer, Heidelberg, Germany.

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

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

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

[16] 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. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018.

[17] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: An optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[18] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`.

[19] Galois system, 2019.

[20] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of The 33rd International Conference on Machine Learning, ICML*, 2016.

[21] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[22] Protocol buffer. `https://developers.google.com/protocol-buffers`. Google Inc.

[23] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[24] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. 2017.

[25] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. `https://arxiv.org/abs/1602.07360`.

[26] Cryptography Lab in Seoul National University. Homomorphic encryption for arithmetic of approximate numbers (heaan). `https://github.com/snucrypto/HEAAN`.

[27] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1209–1222, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[28] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 1651–1669, Baltimore, MD, USA, August 15–17, 2018. USENIX Association.

[29] Alex Krizhevsky. The CIFAR-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`, 2009.

[30] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[31] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 619–631, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[32] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[33] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 35–52, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[34] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[35] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[36] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim,

Javier López, and Taesoo Kim, editors, *ASIACCS 18: 13th ACM Symposium on Information, Computer and Communications Security*, pages 707–721, Incheon, Republic of Korea, April 2–6, 2018. ACM Press.

[37] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 2:1–2:6, New York, NY, USA, 2018. ACM.

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

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

[40] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019(3):26–49, July 2019.

[41] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.