EVA Improved: Compiler and Extension Library for CKKS

Sangeeta Chowdhary[1], Wei Dai[2], Kim Laine[2], and Olli Saarikivi[2]

[1] Rutgers University, New Brunswick, NJ, USA
[2] Microsoft Research, Redmond, WA, USA

Abstract. Homomorphic encryption (HE), especially the CKKS scheme, can be extremely challenging to use. The EVA language and compiler (Dathathri *et al.*, PLDI 2020) was an attempt at addressing this challenge. EVA allows a developer to express their encrypted computation in a simple form with a Python-integrated language called PyEVA. It then compiles the program into an executable form by inserting operations such as relinearization and rescaling, applying optimizations, and choosing encryption parameters with the objective of minimizing execution time. Compiled programs can be executed with a parallelizing back-end against a library of HE primitives.
Our work improves upon the EVA toolchain in several ways: changes to the Python front-end make writing PyEVA programs more natural, while a rework of EVA's C++ APIs makes writing new passes easier. We also implement two new optimizations, common subexpression elimination and reduction balancing, which we show allow users to write simpler and more modular PyEVA programs.
We argue that the abstraction EVA provides is insufficient to resolve some common usability challenges. For example, managing vectors of arbitrary size is non-trivial. To resolve these problems, we demonstrate how building a library of commonly used data structures and functions is simple in PyEVA. EVA's automation allows writing very concise code, which gets fused and optimized together with the user program. We create the beginnings of an EVA Extension Library (EXL), that provides vector and matrix classes and a collection of common statistical functions, to demonstrate the power of this approach.

1 Introduction and Background

We will first discuss homomorphic encryption in general and details of the CKKS scheme both to give background and to motivate the need for a compiler like EVA. Then we will introduce EVA both from a user's and developer's point of view to motivate our contributions.

1.1 Homomorphic Encryption

Homomorphic encryption [33, 23] refers to encryption that allows computation to be done on encrypted data, without requiring secret key material. Modern *fully* homomorphic encryption schemes [7, 6, 22, 24, 20, 14, 12] support at least two different arithmetic or binary operations on encrypted data. In this work we focus exclusively on the CKKS scheme introduced in [12], which we will describe in the next section.

Since 2010 several homomorphic encryption libraries have been developed, implementing many of the schemes mentioned above. These libraries provide only low-level homomorphic encryption primitives, including key generation, encryption, decryption, a few scheme-dependent computational operations on encrypted data, and a variety of "maintenance" operations that are required for the functionality of the scheme. One problem is that homomorphic encryption schemes are generally very sensitive to the computation being organized in the exactly right way, and the maintenance operations must be used appropriately. Another problem is that the encryption schemes have a number of parameters that must be carefully chosen; otherwise, the result of the encrypted computation may be incorrect, or the performance may be several orders of magnitude worse than it needs to be. These details can be difficult to handle even for experts and make homomorphic encryption nearly inaccessible to most developers.

One approach to addressing these issues is to create a domain specific language (DSL) that is tailored for homomorphic encryption and optimizing compiler that reorders the computation according to some heuristic strategy, inserts necessary maintenance operations, selects optimal encryption parameters, and compiles the DSL into native homomorphic encryption library API calls. Several such compilers have been created recently, targeting different use-cases, schemes, and library back-ends [17, 19, 5, 4, 18, 15]. In this work we improve upon the EVA compiler [18] that defines a general purpose DSL for encrypted computations with the CKKS scheme, targeting the Microsoft SEAL [34] library back-end.

1.2 CKKS Scheme

The CKKS scheme [12, 11] was invented in 2016 and quickly gained significant popularity as one of the most useful homomorphic encryption schemes. It is now implemented in several libraries [36, 34, 31, 29]. We refer readers to [11] for full details on the scheme and will limit our discussion here to the parts that are essential for understanding the functionality.

Encryption Parameters The CKKS scheme is parameterized by a power-of-two integer n and an integer $q = \prod q_i$, where q_i are distinct prime numbers congruent to 1 modulo $2n$. In modern implementations the q_i are usually up to around 60 bits in size, but may be much smaller as well. These parameters determine the security level of the scheme as described in [1]. In short, a larger n increases the security level, whereas a larger q decreases the security level. However, a larger q enables computations with higher multiplicative depth to be computed on encrypted data. The number of primes in q does not matter for security, but it does matter for the multiplicative depth. Let $\mathcal{P} = (n, \prod_{i=1}^{k} q_i)$ denote a parameter set for CKKS. We expand $\mathcal{P}$ to derive a chain of parameter sets $\mathcal{P}_0 \to \mathcal{P}_1 \to \cdots \to \mathcal{P}_{k'-1}$, where $\mathcal{P}_j = (n, \prod_{i=1}^{k-j} q_i)$ and $k' \leq k$. Note that $\mathcal{P}_0 = \mathcal{P}$. We would like to emphasize that q is an *ordered product* of (distinct) prime numbers q_i: the order matters a lot, as we will see soon.

We refer to the position of the parameter set used for a specific ciphertext or plaintext as its level. A ciphertext or plaintext using the parameter set $\mathcal{P}_0$ (resp., $\mathcal{P}_{k'-1}$) is said to be at the *highest level* (resp., *lowest level*). We will see below that it is easy to move ciphertexts and plaintexts down in parameter chain until they reach the lowest level. Moving up in the level is also possible, but much more challenging and is less commonly implemented or used [10, 9, 28].

Encoding and Encryption Before encrypting, data must first be encoded into CKKS plaintexts. The encoding process takes a parameter set $\mathcal{P}$ and a scale $s \in \mathbb{R}_{>0}$ and encodes an $n/2$-dimensional vector of complex numbers into a $(\mathcal{P}, s)$-plaintext $\mathtt{pt}_{\mathcal{P},s}$. The encoding process involves permutation, conjugation, and linear transformation on the complex vector, resulting in an output n-dimensional vector of real numbers, which are further multiplied with s and rounded to $\mathbb{Z}$. Thus, the scale quite literally denotes the precision of the fractional part. Encryption converts $\mathtt{pt}_{\mathcal{P},s}$ into a $(\mathcal{P}, s)$-ciphertext $\mathtt{ct}_{\mathcal{P},s}$. There is no way to encode fewer than $n/2$ numbers into a $(\mathcal{P}, -)$-plaintext; consequently, a $(\mathcal{P}, -)$-ciphertext cannot encrypt fewer than $n/2$ numbers.

Changing the Level Let $C_{\mathcal{P},s}$ and $P_{\mathcal{P},s}$ denote the $(\mathcal{P}, s)$-ciphertext and plaintext spaces, respectively. A parameter chain $\mathcal{P}_0 \to \mathcal{P}_1 \to \cdots \to \mathcal{P}_{k'-1}$ induces two operations between these spaces:

$$\textsc{ModSwitch} : C_{\mathcal{P}_j,s} \to C_{\mathcal{P}_{j+1},s}\,, P_{\mathcal{P}_j,s} \to P_{\mathcal{P}_{j+1},s}$$
$$\textsc{Rescale} : C_{\mathcal{P}_j,s} \to C_{\mathcal{P}_{j+1},s/q_{k-j}}\,, P_{\mathcal{P}_j,s} \to P_{\mathcal{P}_{j+1},s/q_{k-j}}$$

In other words, modulus switching (ModSwitch) moves a ciphertext or plaintext one step down in level, without changing the scale, whereas rescaling (Rescale) moves a ciphertext or plaintext one step down in level and divides its scale by the prime number that was removed in the step $\mathcal{P}_j \to \mathcal{P}_{j+1}$ from the modulus q.

We note that encoding, encryption, and these operations do not commute in the mathematical sense as one might expect, but instead encryption is randomized and the other operations introduce various small amounts of error into the plaintexts and ciphertexts.

Encrypted Computing The CKKS scheme supports encrypted element-wise vector addition, negation, multiplication, and complex conjugation, as well as cyclic vector rotation (in either direction); for addition and multiplication it supports both ciphertext-ciphertext and ciphertext-plaintext variants of the operations:

$$\begin{aligned}
\textsc{Add} &: C_{\mathcal{P}_j,s} \times C_{\mathcal{P}_j,s} \to C_{\mathcal{P}_j,s}\,, C_{\mathcal{P}_j,s} \times P_{\mathcal{P}_j,s} \to C_{\mathcal{P}_j,s} \\
\textsc{Neg} &: C_{\mathcal{P}_j,s} \to C_{\mathcal{P}_j,s} \\
\textsc{Mul} &: C_{\mathcal{P}_j,s_1} \times C_{\mathcal{P}_j,s_2} \to C_{\mathcal{P}_j,s_1 s_2}\,, C_{\mathcal{P}_j,s_1} \times P_{\mathcal{P}_j,s_2} \to C_{\mathcal{P}_j,s_1 s_2} \\
\textsc{Conj} &: C_{\mathcal{P}_j,s} \to C_{\mathcal{P}_j,s} \\
\textsc{Rot} &: C_{\mathcal{P}_j,s} \times \mathbb{Z} \to C_{\mathcal{P}_j,s}
\end{aligned}$$

As expected, addition requires both inputs to have the same scale, whereas multiplication tolerates different input scales and yields a result with scale that is the product of the input scales. For all binary operations both inputs must be at the same level.

There is one aspect of ciphertexts and plaintexts that we have ignored thus far: *size*. Concretely, each CKKS $(\mathcal{P}, -)$-plaintext is a single polynomial of degree at most $n-1$ and coefficients modulo q, where $\mathcal{P} = (n, q)$; we say that the size of a plaintext is 1. Each freshly encrypted ciphertext consists instead of 2 such polynomials; thus, we say that the size of such a ciphertext is 2. In practice, addition, negation, ciphertext-plaintext multiplication, complex conjugation, and rotations all preserve the size. If addition inputs two ciphertext of different sizes, the output will have a size that is the larger of the input sizes. However, ciphertext-ciphertext multiplication increases the size as follows: if the inputs have sizes A and B, the output has size $A + B - 1$. By far the most common case is multiplying two ciphertexts of size 2, resulting in an output ciphertext of size 3. Unfortunately, larger ciphertexts are very slow to operate on. To resolve this issue, CKKS supports a so-called relinarization operation, that reduces the size of a ciphertext. In practice this always means reducing from size 3 back to size 2, without changing the underlying plaintext data:

$$\textsc{Relin} : C_{\mathcal{P}_j,s} \to C_{\mathcal{P}_j,s}\,.$$

Stabilizing the Scale We define one more operation for CKKS: a manual scale change. This is a no-op that merely changes the metadata of a ciphertext or plaintext:

$$\textsc{ChangeScale}_{s'} : C_{\mathcal{P}_j,s} \to C_{\mathcal{P}_j,s'}\,.$$

Namely, given a $(\mathcal{P}, s)$-plaintext, one can simply lie to the decoder operation that the plaintext has scale s' instead. When the decoder divides the result vector (of complex numbers) by the scale, it incorrectly divides it by s' instead of s, changing the result by a multiplicative factor s/s' from what it was supposed to be. For example, given a positive real number $r \in \mathbb{R}_{>0}$, one could implement a division-by-r operation by simply changing the scale of a ciphertext or plaintext from initial scale

s to rs. In general this is not very practical or useful, but there is one very important application that we now describe.

Recall that all prime factors q_i of the modulus q are distinct. For example, consider a simple computation like $x^2 + y$ on encrypted inputs x, y, both encoded (and encrypted) at the highest level with the same scale s. Computing x^2 changes the scale to s^2 and maintains the same level; one may want to relinearize at this point, which does not affect the scale or level. But how can we add y to the result? One option would be to change the scale of y up: this is possible, for example, by encoding a vector of all ones with a scale s, and performing a ciphertext-plaintext multiplication with this plaintext. This approach has its uses, but the whole idea of the CKKS scheme is to enable removing of unnecessary precision through rescaling, not adding more of it. Consider instead rescaling the x^2 result, yielding a $(\mathcal{P}_1, s^2/q_k)$-ciphertext, where q_k is some prime. Since y is a $(\mathcal{P}_0, s)$-ciphertext, we are far from being able to add these. We can modulus switch y down one level, yielding a $(\mathcal{P}_1, s)$-ciphertext, but we still cannot add, as $s \neq s^2/q_k$. However, if $q_k \approx s$, then we could simply change the scale of the x^2 ciphertext to s, introducing a multiplicative scaling error $s/q_k \approx 1$, but allowing us to immediately compute the sum. In practice it is convenient to choose the prime numbers q_i to be close to powers of two. For example, suppose every prime q_i is chosen to be close to 2^{40}; then we can encode all input data at scale exactly 2^{40} and the above technique results in the scale stabilizing at 2^{40} in multiplications. The forced scale changes result in a slight error in the result due to the primes q_i only being close to 2^{40}, but this is often insignificant and a small price to pay for the added convenience.

Scale stabilization makes it often convenient to rescale after every ciphertext-plaintext multiplication, and relinearize and rescale after every ciphertext-ciphertext multiplication, although there are many situations where this naive rule results in much worse performance than more clever strategies. We would like to note that in some implementations the order of relinearization and rescaling may matter, but in SEAL it does not.

We would also like to note that many implementations, including SEAL, requires the user to designate one or more of the primes in q as *special primes* that do not take part in the parameter chain construction, but are used for other purposes. While EVA needs to know how to deal with them, special primes do not play any particular role in this work, and we will ignore them in the rest of the text for the sake of simplicity.

Error and Overflow The CKKS scheme preserves approximate computation, producing results with errors in the least significant bits. One minor source of error, machine error in IEEE double precision floats, is introduced when the input complex vector is linearly transformed to a vector of real numbers in encoding (Section 1.2) and when the reverse transformation is performed in decoding. This is the maximum effective precision that can be achieved in most concrete implementations. Another more important source of error is the so-called *noise* added to the least significant bits of encrypted data by all CKKS operations. To be precise, the CKKS scheme preserves approximate computation on a transformed vector of real or complex numbers, whose fractional precision is initially controlled by the user in choosing the scale. The exact fractional precision preserved lives between the noise and the scale, so if the scale is large enough, the relevant part of the data remains protected from the noise. In practice, one may want to test whether specific choices for scale in the inputs yields appropriate precision.

The underlying data in ciphertexts are real values multiplied by a scale s and rounded to integers modulo q. If the integer part of the underlying data is too large it exceeds the capacity of integers modulo q, causing an overflow situation. This typically wipes out the entire result. Overflow is rarely a problem in the encoding process, but can become a problem at lower levels

(i.e., deeper into the encrypted computation). To avoid this problem, the programmer must ensure that throughout the computation the data always fits into integers modulo *the current* q. On one hand, encrypted computations often increase the underlying data. On the other hand, rescaling reduces the size of q. Thus, the user must ensure through proper choices of the scale, q, and the layout of the computation, that the data will not overflow, while maintaining a large enough scale for a high-enough effective precision.

Designing CKKS Programs At this point the reader might already anticipate the challenge of designing CKKS programs. For a complicated program with multiple inputs and outputs it is incredibly difficult to find a good set of parameters that provides sufficient accuracy and efficiency, and yields consistency with the limitations of the various operations described above. It is difficult to appropriately balance the scales and levels for the internal "wires" in a computation. Moreover, changing the input scale requires often a total change in the encryption parameters and possibly changes to how the computation is structured, making it difficult for a programmer to test whether specific settings would yield better accuracy or performance than other settings. These are the problems the EVA language and compiler [18] were designed to solve.

1.3 EVA

To address the usability issues of CKKS, Dathathri *et al.* [18] presented a domain-specific language and an optimizing compiler targeting the CKKS scheme. EVA has a Python front-end, PyEVA, that allows computations to be expressed using basic arithmetic operations in Python, creates a computation graph, inserts appropriate rescaling (and other) operations, and uses heuristic approaches to find good encryption parameters.

We have made several changes to EVA to improve usability for both users and developers. We present EVA here in its improved form and detail our changes in Section 2.

PyEVA The EVA toolchain creates several types of objects: a *program* (input or compiled), *encryption parameters*, a *signature*, a *public context*, and a *secret context*. The input program is created by the user using the PyEVA language. Such a program may look as follows:

```
from eva import EvaProgram, Input, Output
prog = EvaProgram('example', vec_size=4)
with prog:
    inx = Input('x')
    iny = Input('y')
    sqsum = inx**2 + iny**2
    Output('out', sqsum + inx + iny)
prog.set_input_scales(30)
prog.set_output_ranges(20)
```

The above listing creates an EVA program computing the squared sum of two inputs plus the sum of the two inputs. Each value is a vector (encrypted) of size 4, and the program is evaluated in a SIMD fashion on the input vector values. Due to the limitations of how CKKS must be parameterized, the vector size must be a power of two. We set the scale for the program to be 30, which means EVA will attempt to stabilize the scales at 2^{30} (Section 1.2) and indicate that the absolute value of all outputs is at most 2^{20}, which EVA uses to avoid overflow.

The next step is to compile the input program into a fully functioning EVA program that can be executed on encrypted data. For example, the above program is in no way ready to be executed: there is no information about what encryption parameters to use, where to perform modulus switching or rescaling, and where to relinearize. Compilation handles all this:

```
from eva.ckks import CKKSCompiler
compiler = CKKSCompiler()
prog, params, signature = compiler.compile(prog)
```

EVA inserts the necessary maintenance operations into the computation and selects appropriate encryption parameters. In addition to the compiled program, EVA returns an object containing the encryption parameters, as well as a signature object describing how each input should be encoded.

Suppose you want to encrypt private input data and share it with an evaluator, e.g., an untrusted cloud computing environment, holding the program object. The evaluator would share the parameter and signature objects with you, allowing you to set up a public/secret key pair, and encrypt your input data:

```
from eva.seal import generate_keys
public_ctx, secret_ctx = generate_keys(params)
input = {'x': [1,2,3,4], 'y': [5,6,7,8]}
enc_input = public_ctx.encrypt(input, signature)
```

The public context holds only public key data and can in principle shared with other parties for providing input data to the computation; the secret context holds the secret key and should be kept private. The input data is provided as a dictionary keyed by the input wire names. For each input wire, we provide a list of size 4 – the vector size specified when the program object was created.

Next the encrypted inputs and the public context must be sent to the evaluator, who can then execute the EVA program:

```
enc_output = public_ctx.execute(prog, enc_input)
```

The encrypted outputs can be sent back to the secret context owner, who can decrypt them and obtain the results:

```
output = secret_ctx.decrypt(enc_output, signature)
```

The decrypted result is a dictionary, keyed by names of the outputs. In this case we had only a single output wire named `out`. Printing `outputs['out']` returns a list of size 4.

Internal Representation (IR) EVA has a unified IR for input programs, all stages of compilation and executable format. This IR has an in-memory representation implemented in C++ on which the compiler's passes operate on. Additionally, there is a serialized representation implemented with Protocol Buffers for use as a wire and on-disk format.

EVA programs are directed acyclic graphs (DAGs) of terms, where a term has and maintains:

`op` The opcode that determines the computation performed by this term, one of those in Table 1.
`operands` List of all terms used as operands for this term.
`uses` List of all terms that include this term in their operands.

Each term is additionally categorized into one of three types: `Cipher`, `Plain` and `Raw` (new; see Section 1.2). Types are deduced by EVA at compile time.

In addition to the terms, each EVA program has and maintains:

`name` User-given name for the program.
`sources` and `sinks` The terms that have no operands and uses, respectively. These are used as entry points for forward and backward traversals.
`inputs` and `outputs` Maps from names to `Input` and `Output` terms, respectively.
`vec_size` Global size for all EVA vectors. Independent of the SEAL vector size, i.e., the number of slots.

Opcode	Description
`Input`, `Output`	Markers for input/output values.
`Constant`	Produce a Raw value.
`Negate`	Unary negation.
`Add`, `Sub`, `Mul`	Binary arithmetic operations.
`RotateLeftConst`	*New:* Rotate left by a fixed offset.
`RotateRightConst`	*New:* Rotate right by a fixed offset.
`Relinearize`	Relinearize result of multiplication.
`ModSwitch`	Drop next modulus.
`Rescale`	Use next modulus to divide value and scale.
`Undef`	Illegal in valid programs. Internal use only.
`Encode`	*New:* Encode `Raw` value into `Plain`.

Table 1. Operations supported by EVA, including *new* ones introduced in this work. Opcodes above the line are the ones user programs may use.

Passes EVA operates on this IR with two kinds of program traversals: *rewrite passes* are allowed to modify, create and remove terms and are always single-threaded, while *analysis passes* only visit each term, but may use a highly scalable multi-core traversal implemented with the Galois library [30]. Both kinds of passes may additionally use a *forward* or *backward* traversal, which guarantee that each terms operands or uses, respectively, are visited before term itself.

Pass	Description
`CommonSubexpressionEliminator`	*New:* Combine duplicate terms.
`TypeDeducer`	Populate map from terms to types.
`ConstantFolder`	Replace terms with only `Constant` operands with a `Constant` term.
`ReductionCombiner`	*New:* Flatten trees of `Add` or `Mul` terms into N-ary terms.
`ReductionLogExpander`	*New:* Expand N-ary terms

`MinimumRescaler`
`AlwaysRescaler`
`EagerWaterlineRescaler`
	} Rescaling policies; see [18].

`LazyWaterlineRescaler`	Similar to `Eager` version, but delay rescaling until `Mul` terms.
`EncodeInserter`	*New:* Add `Encode` terms for `Raw` operands of `Cipher` terms.
`EagerRelinearizer`	Add `Relinearize` after each `Mul`.
`LazyRelinearizer`	Add `Relinearize` after each `Mul` as late as possible.
`ModSwitcher`	Add `ModSwitch` terms to bring unequal operands to same level.
`SEALLowering`	Do SEAL specific transformations.
`LevelsChecker`	Assert all operand levels match.
`ParameterChecker`	Check `Rescale` consistency.[3]
`ScalesChecker`	Check `Add` and `Sub` terms operands have equal scale.
`EncryptionParametersSelector`	Select CKKS parameters based on `Rescale` terms.
`RotationKeysSelector`	Find all rotation offsets required.

Table 2. Passes implemented in EVA. Listed in order of use in EVA's CKKS compiler, although all are not always used.

Table 2 lists both existing passes in EVA and passes added in this work. Most passes use a forward traversal, with `ModSwitcher` being the only example of a backward pass.

[3] The Minimum and Always policies can't handle some programs, in which case this pass will instruct the user to switch policies.

Back-end EVA includes a back-end for executing compiled programs against SEAL's CKKS implementation. This is implemented using a forward analysis pass that constructs a map from terms to values, with optional use of the multi-core support for parallelization. Mappings for `Input` terms are initialized with inputs values provided by the user and after the pass has run output is produced from the values stored for any `Output` terms.

Limitations PyEVA allows the user to specify whether `Input` terms are encrypted or not. For example, a computation may involve data from multiple data owners, one of which hopes to input encrypted data, and the other unencrypted data. However, EVA did not implement support for computation between unencrypted values, as SEAL does not implement pure plaintext operations. Section 2.4 describes our changes to enable these scenarios.

1.4 Our Contributions

Our contributions are two-fold. First, we improve the EVA toolchain in multiple ways:

- PyEVA now has: (1) transparent support for Python numbers and lists; (2) named inputs and outputs instead of positional; (3) scaling specified outside program (Section 2.1).
- A rewrite of EVA's internal APIs make it easier for developers to extend with new passes (Section 2.2).
- A new `Raw` type and `Encode` operation allow unencrypted inputs to be used on multiple levels without having to perform modulus switching on SEAL plaintexts or transmit multiple encodings of the same input (Section 2.4).
- Two new optimizations, common subexpression elimination (CSE) and reduction balancing, allow simpler and more modular PyEVA programs (Sections 2.5 and 2.6).

Second, we note that despite our improvements to PyEVA, many common computations can still be relatively tedious to write due to its low level of abstraction. As one example, summing the elements of a ciphertext (i.e., "horizontal sum") is a very standard operation, but is non-trivial to implement.[4] As another example, basic vector and matrix arithmetic can be surprisingly complicated to implement with EVA due to the limitations in the vector sizes that the SEAL back-end supports.[5]

We propose an *EVA Extension Library* (EXL) containing such common functions implemented in PyEVA. The benefits of implementing EXL in PyEVA, instead of directly against SEAL, are: (1) it is much easier to do; (2) EVA can co-optimize EXL functions together with the rest of the program. We create the beginnings of EXL, including the horizontal sum function, simple vector and matrix classes, and a few statistical functions, to demonstrate the power of this approach (Section 3).

1.5 Related Work

Multiple homomorphic encryption compiler projects have been done in the past, targeting different schemes, different library back-ends, and different use-cases [37]. Some of the projects clearly define a custom DSL and an obvious optimizing compiler component, whereas others could more correctly be called libraries, wrapping the hard-to-use homomorphic encryption libraries with a more convenient

[4] A version with alternating power-of-two rotations and addition requires $O(\log n)$ operations while the naïve sum of all rotations needs $O(n)$.

[5] This is not exactly a limitation of SEAL, but rather a limitation of the algebraic structures CKKS is based on.

interface targeting specific applications. Most of the projects are incomparable with each other: they target different scenarios or back-ends with vastly different properties.

One of the first attempts at a homomorphic encryption was created for the IARPA RAMPARTS program [2]. This compiler allowed the programmer to write computations in the Julia language, and translated them to appropriate PALISADE [31] API calls.

Alchemy [17] compiles for the $\Lambda \circ \lambda$ library [16], which implements a BGV scheme variant. A valid program written in the Alchemy DSL is guaranteed to work correctly and minimize the chance of runtime errors. This is an important property for compilers, because mistakes in homomorphic encryption programs written using native library APIs, or improper parameterizations, will typically lead to runtime errors that can be difficult to debug for non-experts.

Some compilers even support multiple back-ends. Cingulata [8] (earlier known as Armadillo) allows the user to write programs in C++ and translates them into Boolean or arithmetic circuits that can be executed using the TFHE library (Boolean circuits) or a custom implementation of BFV (arithmetic circuits). SHEEP [35] is possibly the most generic of all compiler projects: it defines a DSL that can be compiled to multiple library back-ends and schemes. Marble [38] is to some extent similar in nature to Cingulata. E3 [13] is also designed for multiple back-ends.

Porcupine [15] specifically targets the difficulty in manually vectorizing programs. As was explained earlier, homomorphic encryption often operates on large encrypted vectors in a SIMD fashion. However, producing code that leverages this parallelism is not straightforward for developers; this is what Porcupine helps with. It compiles a custom DSL to the BFV scheme API in SEAL.

CHET [19], nGraph-HE [5, 4], SEALion [21], and TenSEAL [3] target machine learning applications by compiling primarily to the CKKS scheme.

Most recently Google announced the homomorphic encryption transpiler [25] targeting the TFHE scheme. The transpiler allows the user to write normal C++ functions and repurposes the XLS toolchain to translate the functions to TFHE API (Boolean circuits), instead of Verilog code.

As has been explained above, EVA [18] defines a DSL and compiles to the CKKS scheme API in SEAL. The present work improves upon EVA.

2 Improved EVA

This section describes and motivates the improvements we have made to EVA. We have also made EVA available open-source under the MIT license to engage the homomorphic encryption users and research communities.[6]

2.1 PyEVA Front-End Improvements

We have made several improvements to PyEVA that make writing programs more natural. Figure 1 shows Sobel edge detection implemented with the improved PyEVA language. We invite the reader to compare this with Figure 6 in [18], which presented the same program in the original PyEVA language.

PyEVA now allows using Python numbers and lists of numbers transparently in expressions involving encrypted values. When a Python value is encountered, EVA automatically creates a Constant term. In contrast, the old PyEVA required the user to call a `constant(scale,value)` function to mark constants and indicate their scale. A consequence of this is that the user no longer has to set the scales of constants, and instead EVA's new `Raw` type support described in Section 2.4 infers the required scales.

[6] `https://GitHub.com/Microsoft/EVA`

```
def convolution(image, width, kernel):
    conv = 0
    for i in range(len(kernel)):
        for j in range(len(kernel[0])):
            rot = image << i * width + j
            conv += rot * kernel[i][j]
    return conv

def sqrt(x):
    return 2.214*x - 1.098*x**2 + 0.173*x**3

sobel = EvaProgram("sobel", vec_size=64*64)
with sobel:
    image = Input("image")
    hor = convolution(image, 64,
        [[-1,0,1], [-2,0,2], [-1,0,1]])
    ver = convolution(image, 64,
        [[-1,-2,-1], [0,0,0], [1,2,1]])
    Output("image", sqrt(hor**2 + ver**2))
```

Fig. 1. Sobel edge detection in PyEVA

The new PyEVA omits scales in the `Input` and `Output` functions. Instead, the user calls `set_input_scales` and `set_output_ranges` to set these parameters. This allows the orthogonal concern of setting appropriate scales to be separated from specifying the computation.

EVA and PyEVA now use named arguments instead of positional arguments. This makes it easier to rearrange the computation inside a PyEVA program without having to change application code that handles inputs and outputs.

2.2 Making Passes Easy to Write

A primary goal of EVA, alongside making homomorphic encryption accessible to non-experts, is to make it easy for homomorphic encryption experts to translate their knowledge into executable form. By implementing passes that achieve the optimizations they desire, their knowledge can benefit every new application and user. From this point of view, it is critical that the internal APIs of EVA are expressive and ergonomic to use.

Dealing with loops in transformations and analyses is in our view one of the main challenges in compilers for beginners. The original EVA [18] took the most important step in making passes easy to write by designing the IR as a loop-free DAG. Instead of having to think about lattices, fixpoints, and termination, developers of new passes can focus on the logic of their transformations inside simpler forward and backward traversals. The tradeoff is that the program representation is less compact, but in our experience this has not been a problem as due to the relative slow-down of homomorphic encryption programs tend to be smaller than traditional ones.

We have continued the work started in the original EVA by further refining its APIs to make writing passes easy. This section details the most important improvements.

Rewriting API We have analyzed the API usage patterns in existing passes and identified the most common transformations made to the DAG. While passes in the original EVA operated directly on C++ `vectors` for both operands and uses, we have encapsulated common modifications as the following new member functions in EVA's `Term` class:

```
void addOperand(const Ptr &term);
bool eraseOperand(const Ptr &term);
bool replaceOperand(Ptr oldTerm, Ptr newTerm);
void setOperands(std::vector<Ptr> o);
void replaceUsesWithIf(Ptr term,
        std::function<bool(const Ptr &)>);
void replaceAllUsesWith(Ptr term);
void replaceOtherUsesWith(Ptr term);
```

The `replace*UsesWith*` functions significantly simplified existing passes in EVA by replacing complex loops over the uses of terms.

Original EVA required passes to manually keep the pointers from terms to their uses up to date, which was a common source of bugs. Our changes remove direct write access to the operands list and instead have the member functions of terms automatically manage the use list. This significantly simplified code in passes.

The `Ptr` type above is an `std::shared_ptr`, which makes memory management simple. The original EVA had chosen to use `shared_ptr` for use pointers and `weak_ptr` for operands, which did not matter when uses were manually updated. However, with automatic use management switching the direction of ownership gave us *dead code elimination* for free: any term that does not appear in a subexpression of an `Output` term is automatically freed.

Attribute	Type	Used in
RescaleDivisorAttribute	uint32_t	`Rescale`
RotationAttribute	int32_t	`Rotate*Const`
ConstantValueAttribute	shared_ptr¡Constant¿	`Constant`
TypeAttribute	Type	`Input`
RangeAttribute	uint32_t	`Output`
EncodeAtScaleAttribute	uint32_t	`Input`,`Encode`
EncodeAtLevelAttribute	uint32_t	`Input`,`Encode`

Table 3. Attributes currently in EVA.

Attributes To improve EVA's API ergonomics we have introduced *attributes*, which are strongly typed named constants attached to terms. Attributes solve several problems in EVA:

- Named and strongly typed attributes make passes more readable and safer by avoiding manual type checks.
- EVA's type system for terms can be kept simple and targeted to homomorphic encryption by moving complex metadata into attributes.
- As EVA adds support for new schemes, back-ends and optimizations, adding information as fields in terms would inflate the memory footprint for all terms and during all passes. Attributes allow only storing what is required.

Figure 3 lists the attributes currently included and which terms they are used in. Attributes are accessed with templated `has<A>`, `get<A>` and `set<A>` methods directly from the `Term` class, where `A` is a tag type naming the attribute. Terms store their attributes in a linked list that embeds the first element in the term instance, which for most cases avoids an indirection.

Term Maps While attributes are suitable for metadata that should be persisted, many passes require tracking more ephemeral per-term information. For example, EVA's rescaling policies track the scale of all terms, but this is only required for at execution time for `Input` and `Encode` terms. For such usage we provide two template classes: `TermMap<T>` and `TermMapOptional<T>`. These encapsulate a contiguous array of type `T` that can be indexed directly with term instances. Internally EVA maintains a unique index for each term in a program and automatically manages space in any registered term maps when new terms are created.

Term maps provide a safe replacement for EVA's previous ad-hoc usage of `std::vector` with a globally incremented index per term. Hashmaps would have been another option, but EVA's multi-core support would have required a concurrent implementation. Term maps are trivially thread safe when passes write only to the current term, thanks to pre-allocation of the buffer.

2.3 Pseudo-Code

For readability, the following sections present our new passes in pseudocode that closely matches the structure of their C++ counterparts in EVA. Common mathematical notation is used for brevity. The global mappings used in the pseudocode correspond to term maps. A map M is indexed with a term t with $M[t]$ and an uninitialized map is $\emptyset$. A list L can be indexed by a non-negative integer i with $L[i]$, lists are constructed with $[a, b, c, \dots]$ and concatenated with $L +\!\!+ L'$. Attributes are accessed with t.get⟨`AttributeName`⟩, modified with t.set⟨`AttributeName`⟩(), and their names are abbreviated for brevity.

2.4 Raw Type and Encoding Insertion

Many applications need to deal with both encrypted and unencrypted data. For example, machine learning inferencing tasks may operate on encrypted inputs, but allow weights to be unencrypted.

EVA's previous approach to unencrypted inputs was to perform CKKS encoding for all input values, but skip the encryption step for unencrypted inputs. This, however, meant that arithmetic between unencrypted values was not supported, as SEAL does not offer arithmetic for plaintext values and, while implementable, it would not be desirable either due to the slowdown involved. While it is always possible to move these kinds of computations to the surrounding application code, allowing unencrypted arithmetic inside EVA programs is more flexible. Another issue with performing encoding early is that this inflates the size of those inputs and may result in higher communication costs.

We have extended EVA's type system with a new `Raw` type alongside the existing `Cipher` and `Plain` types. Values of type `Raw` represent vectors of `vec_size` double precision floating point elements. All `Constant` terms and Inputs that the user has marked unencrypted (by passing `is_unencrypted=True` to `Input`) are of type `Raw`. All of EVA's arithmetic operations are supported between terms of type `Raw`, while operations specific to homomorphic encryption, such as `ModSwitch`, naturally are not.

To move values of type `Raw` to `Plain`, we've added a new opcode, `Encode`, which has a single operand for the `Raw` value to be encoded and two attributes, `Scale` and `Level`, to control how the value gets encoded. `Encode` terms are added by a new Encoding Insertion pass detailed in Algorithm 1. It depends on two prepopulated maps: S for the scales and T for types of all existing terms, which are produced by EVA's rescaling insertion and type deduction passes, respectively. For each term of type `Cipher`, the pass checks if any of the operands are of type `Raw` and if so inserts new `Encode` terms.

Require: S and T are maps from existing terms to their scales and types, respectively, and VisitEI is used in a forward traversal.
Ensure: There are no terms with mixed `Cipher` and `Raw` operands.

```
 1: procedure VisitEI(t)
 2:     if T[t] = Cipher then
 3:         for o ∈ t.operands s.t. T[o] = Raw do
 4:             e ← NewTerm(Encode, [o])
 5:             if t.op = Add ∨ t.op = Sub then
 6:                 e.set⟨Scale⟩(S[t])
 7:             else
 8:                 e.set⟨Scale⟩(S[o])
 9:             T[e] ← Plain
10:             S[e] ← e.get⟨Scale⟩
11:             t.replaceOperand(o, e)
```

Algorithm 1: Encoding Insertion

One complication handled by the pass is that for `Add` and `Sub` all operands must have the same scale and thus the pass sets the `Scale` attribute appropriately. For multiplications the scale set for the operand is used, which is typically one set by the user in PyEVA with `set_input_scales`. `Encode` terms also need the `Level` attribute set, but this is added during EVA's modulus switching insertion pass, which we modified to add support for `Encode`.

2.5 Common Subexpression Elimination

Common Subexpression Elimination (CSE) is a well known optimization that ensures a program has a unique representative term for each syntactically equivalent subexpression. Providing a CSE pass in EVA means that authors of EVA programs do not have to memoize common subexpressions. As an example, consider the program in Figure 1. The program calls the `convolution` function twice to run separate filters for detecting horizontal and vertical edges. However, as both are 3×3 kernels they will perform the same rotations on line 5. While this could be remedied in the user code by memoizing the results of line 5 or by fusing the loops of the two invocations (as was done in Figure 6 of [18]), being able to rely on CSE greatly simplifies the user code. Similarly, on line 10 the polynomial approximation of square root does not have to factor out the shared powers of `x`.

We have implemented CSE in EVA as detailed in Algorithm 2. During a forward pass CSE checks for each term if a syntactically equivalent term r was already visited earlier. If so, all uses of the current term t are redirected to r, by replacing t with r in all operand lists that mention t. If no such r exists, then t is kept and remembered as the representative of its syntactic equivalence class.

SyntacticEquals checks that the terms have the same opcode and operands. Crucially, the operands are compared using pointer equality, avoiding recursive calls into SyntacticEquals. This works, because in a forward pass operands are visited before the term itself. After checking the opcode and operands, operation specific attributes may be checked. For example, in the case of rescaling it is checked that the divisors match. Inputs and outputs are never eliminated, as EVA already ensures there's a single representative for each named input/output. The implementation of CSE uses a hash set for U, which requires hash codes in addition to the equality operation. These syntactic hashes are calculated using the same fields that SyntacticEquals uses.

Require: $U = \emptyset$ and VisitCSE is used in a forward traversal.
Ensure: Program has no two syntactically equivalent terms.

```
 1: procedure VisitCSE(t)
 2:     if ∃r ∈ U : SyntacticEquals(t, r) then
 3:         t.replaceAllUsesWith(r)                          ▷ Remove term t.
 4:     else
 5:         U ← U ∪ {t}
 6: procedure SyntacticEquals(t, r)
 7:     if t.op ≠ r.op ∨ t.operands ≠ r.operands then
 8:         return False
 9:     switch t.op do
10:         case Undef
11:             return False
12:         case Negate, Add, Sub, Mul, Relinearize, ModSwitch
13:             return True
14:         case Input, Output
15:             return t = r                                 ▷ Pointer equality.
16:         case Constant
17:             return t.get⟨ConstValue⟩ = r.get⟨ConstValue⟩
18:         case RotateLeftConst, RotateRightConst
19:             return t.get⟨Rotation⟩ = r.get⟨Rotation⟩
20:         case Rescale
21:             return t.get⟨Divisor⟩ = r.get⟨Divisor⟩
22:         case Encode
23:             return t.get⟨Scale⟩ = r.get⟨Scale⟩ ∧
                    t.get⟨Level⟩ = r.get⟨Level⟩
24:     return False                                         ▷ Unreachable, but it's safe to return false.
```

Algorithm 2: Common Subexpression Elimination

2.6 Reduction Balancing

Consider the following arithmetic expressions:

$$(a \cdot b) \cdot (c \cdot d) \tag{1}$$

$$((a \cdot b) \cdot c) \cdot d \tag{2}$$

While arithmetically equivalent, expression 1 is better because it has a lower multiplicative depth. While it is always possible to avoid the equivalent of expression 2 in PyEVA programs manually, doing so may make code less modular. Consider the following program:

```
def poly(x):
  return 0.837 * x**2

prog = EvaProgram("inbalanced", vec_size=4096)
with prog:
  a = Input("a")
  b = Input("b")
  Output("c", poly(a) * b)
```

The expression constructed for the output `c` is essentially $((x \cdot x) \cdot 0.837) \cdot b$, which has depth 4. Balancing this expression in user code would require inlining the `poly` function by rewriting line 8 to `Output("c", (0.837 * b) * a**2)`. In some cases the exponent operator has to be avoided too, as for example `a * b**3` produces an unbalanced expression.

To remedy these problems we have added a reduction balancing feature to EVA by implementing two passes that run in succession. The first pass, detailed in Algorithm 3, flattens any trees of

Require: VisitRC is used in a forward traversal.
Ensure: Trees of reductions with `Mul` or `Add` are collapsed into single terms with multiple operands.

```
1: procedure VisitRC(t)
2:     if t.op ≠ Mul ∨ t.op ≠ Add then
3:         return
4:     if |t.uses| = 1 then
5:         use ← t.uses[0]
6:         if use.op = t.op then
7:             while use.eraseOperand(t) do
8:                 for o ∈ t.operands do
9:                     use.addOperand(o)
```

Algorithm 3: Reduction Combiner

reductions with either the `Mul` or `Add` operation into a single term with all the source terms to the subtree as operands. Note that terms with multiple uses are always retained even if they are otherwise part of a reduction subtree, since that intermediate result is needed elsewhere.

The second pass, detailed in Algorithm 4, expands the flattened reductions back into balanced binary trees. One subtlety to be considered is that it is beneficial to both reduce `Raw` terms as well as `Cipher` terms of a similar scale together first. For example, if in expression 1 terms a and c are of type `Raw`, while b and d are `Cipher`, then the encoding insertion pass in Algorithm 1 will add two `Encode` terms. However, if the `Raw` terms are a and b instead, only one `Encode` term is required. The sorting on line 6 together with the scale estimation in the SetScale procedure implement a heuristic that groups terms with the same type and scale together.

The balanced reduction trees produced by these passes minimize the amount of modulus consumed by multiplications. The transformation is also useful in the case of `Add` operations as balanced trees expose more parallelism: an unbalanced tree requires $O(n)$ time to evaluate in the worst case, while with sufficient processors a balanced tree only requires $O(\log n)$ time.

3 EVA Extension Library

3.1 Why an Extension Library?

Even though EVA greatly improves the user experience of CKKS, we note that it still leaves much to be desired. We illustrate the problem with a few examples.

Consider computing the sum of all elements in an EVA vector, e.g., when evaluating a high-dimensional dot product, where the EVA program's vector size is set to a large value (several thousands) and a single SIMD multiplication is performed between two input vectors. To complete the dot product, the elements of the multiplied vector must be added up. This can be done using rotations and additions, but a naïve version will use several thousand rotations, while a good one gets by with a logarithmic number of rotations. Ideally, a good implementation for the "horizontal sum" would be readily available to avoid this pitfall.

It is common for applications to include computations that are not directly supported in homomorphic encryption, in which case *polynomial approximation* becomes useful. Given an input function, a Taylor polynomial, a Chebyshev approximation, or some other polynomial approximation valid for a specific input domain may be used. It is very application dependent which kind of approximation is appropriate, so a library of several methods would be useful.

As a third example, consider the case of arbitrary-sized vectors or matrices. Such objects are not simple to implement using the power-of-two size vectors that EVA natively supports. While

Require: $S = \emptyset$, T is a map from existing terms to their types, and VisitRLE is used in a forward traversal.
Ensure: There are no terms with mixed `Cipher` and `Raw` operands.

procedure VisitRLE(t)
  SetScale(t)
  if t.op $\neq$ `Mul` $\vee$ t.op $\neq$ `Add` $\vee$ $|t.\text{operands}| \leq 2$ **then**
    return
  $O \leftarrow t$.operands
  $O \leftarrow$ Sorted($O, \lambda a, b : T[a] = \texttt{Cipher} \Rightarrow T[b] = \texttt{Cipher} \wedge S[a] \leq S[b]$)
  while $|O| > 2$ **do** ▷ Expand O into balanced tree.
    $O' \leftarrow []$
    $i \leftarrow 0$
    while $i + 1 < |O|$ **do**
      $O' \leftarrow O' +\!\!+ [\text{Newterm}(\texttt{t.op}, [O[i], O[i+1]])]$
      $i \leftarrow i + 2$
    if $i < |O|$ **then**
      $O' \leftarrow O' +\!\!+ [O[i]]$
    $O \leftarrow O'$
  assert $|O| = 2$
  t.operands $\leftarrow O$
procedure SetScale(t)
  if $|t.\text{operands}| = 0$ **then**
    $S[t] \leftarrow t.\text{get}\langle\texttt{Scale}\rangle$ ▷ Sources have user given scales.
  else if $T[t] = \texttt{Cipher} \wedge t.\text{op} = \texttt{Mul}$ **then**
    $S[t] \leftarrow \Sigma_{o \in t.\text{operands}} S[o]$
  else
    assert $\forall o, o' \in t.\text{operands} : S[o] = S[o']$
    $S[t] \leftarrow S[t.\text{operands}[0]]$

Algorithm 4: Reduction Log Expander

these could be directly offered in EVA's back-end, especially for matrices there are numerous ways to encode the data and implementing them all would inflate the complexity of EVA's core.

We propose building an *EVA Extension Library* (EXL) in Python using PyEVA to address these use-cases. We argue that such a library would be impractical to build directly using SEAL or other implementations of CKKS, as such low-level implementations would become very complex due to the lack of optimizations and automation that a compiler like EVA provides. Building EXL in PyEVA makes it accessible to a huge group of developers, makes it modular and convenient to use, and makes it performant by allowing EVA to automatically optimize the library functions as they are fused into EVA programs.

3.2 Horizontal Sum

As the first demonstration of how easy EXL is to build in PyEVA, we present an efficient horizontal sum implementation:

```
def horizontal_sum(x):
  i = 1
  while i < len(x):
    y = x << i # rotation by i steps
    x = x + y
    i *= 2
  return x
```

By successively rotating the value by increasing powers-of-two and adding the rotated value back, a horizontal sum can be computed in a logarithmic number of rotations and additions. PyEVA operator overloading makes the code read like normal Python code.

3.3 Vector and Matrix

As a significant concrete demonstration, we implemented an EXL vector class for arbitrary-sized vectors in PyEVA.

SEAL vector and EVA vector The new EXL vector is distinct both from the native SEAL vectors (i.e., ciphertexts and plaintexts) that always have a fixed size set by the encryption parameters (Section 1.2), and from the EVA vectors that still have a power-of-two size determined by the user when creating the EVA program (Section 1.3).

Internally, EVA vectors of size less than the SEAL vector size are implemented by replicating the EVA vector values until the SEAL vector is filled. Large EVA vector sizes are always accommodated by potentially inflating the encryption parameters to make the SEAL vector at least the same size. In a "wide" enough EVA program, the EVA vector size matches the SEAL vector size to avoid redundant computation and unnecessarily large encryption parameters.

EXL vector These issues make some very natural computations unnecessarily complex, as applications often need to operate on data of arbitrary size. Consider an EVA program with a very large input vector, e.g., a million elements. Setting the EVA vector size to match would be very inefficient due to the large encryption parameters required. Instead, the user would have to manually break up their input data into multiple EVA vectors and express the computation in terms of these, but now the user has to also break up their computations to work in terms of these subsections of the data. Rotations, in particular, become very complex due to the number of corner-cases involved.

Our EXL vector class resolves all of these problems; it handles the idiosyncrasies of splitting up user data of any size to fit into EVA vectors of any size. The generated code also naturally benefits from EVA's automatic parallelization. Consider the following example:

```
def vector_horizontal_sum(x):
  x = sum(x.exprs)
  return horizontal_sum(x)

def vector_dot_product(x, y):
  return vector_horizontal_sum(x * y)

prog = EvaProgram('DotProduct', vec_size=4)
with prog:
  v = VecInput('v', 5)
  w = VecInput('w', 5)
  VecOutput('y', vector_dot_product(v, w))
```

The `vector_horizontal_sum` function provides an equivalent to `horizontal_sum` for the EXL vector. The `vector_dot_product` function then implements dot product for two arbitrary-sized vectors in one line of code. The user program can now invoke these functions with EXL vector inputs obtained with the `VecInput` function and produce an output with the `VecOutput` function.

Figure 2 shows how the EXL vector of size 5 is divided into two EVA vectors of size 4, which are replicated into two SEAL vectors of size 4096. Note that in this case the user would benefit from increasing the EVA vector size to 8 or more.

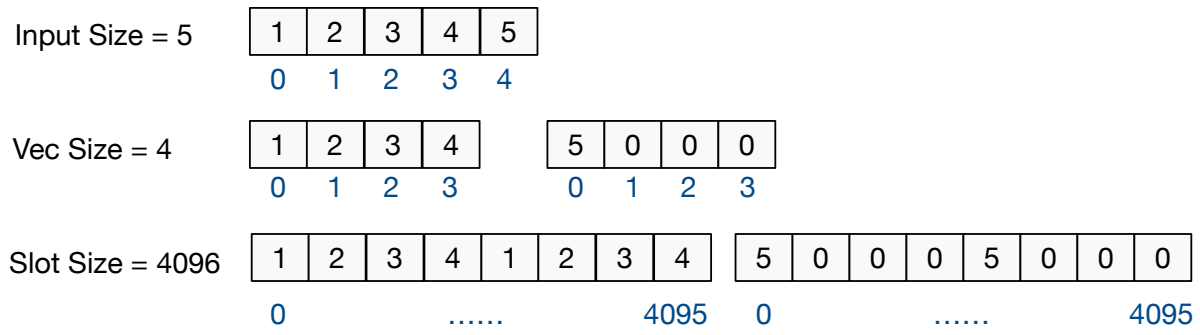

Fig. 2. This graph shows the instance for splitting the input into vectors of size of power of two using EXL Vector class.

Matrix Similarly to the EXL vector, we implement a simple EXL matrix class. Matrix-vector and matrix-matrix products are fundamental operations in many interesting applications, e.g., machine learning. We implemented the matrix-vector product using the well-known trick of splitting the matrix in diagonal order. Suppose $A = (a_{i,j})$ is an $m \times m$ matrix and $v = (v_i)$ is vector of length m. EXL splits A into m vectors in diagonal order:

$$d_1 = \{a_{1,1}, a_{2,2}, \ldots, a_{m,m}\}$$
$$d_i = \{a_{1,i}, a_{2,i+1}, \ldots, a_{m-i+1,m}, a_{m-i+2,1}, \ldots, a_{m,i-1}\}$$

The product Av is computed as: $\sum_{i=1}^{m} d_i \cdot \textsc{Rot}(v, i)$, where function Rot (Section 1.2) cyclically rotates the vector v by i steps.

3.4 Vector Size and Program Generators

EVA partly decouples the sizes of vectors used in input programs from the SEAL vector size, i.e., the number of slots, determined in EVA's encryption parameter selection. If the EVA vector size is smaller than the SEAL vector size, the executor emulates the smaller size by replicating vectors. On the other hand, if it is larger, then the SEAL vector size is expanded to fit the EVA vector size.

While this scheme allows users to select any (power-of-two) vector size for their EVA programs and get a runnable program, it has drawbacks. If the EVA vector size is smaller than the SEAL vector size then each operation is performing redundant computation. On the other hand, if the SEAL vector size was expanded to a larger size to fit the EVA vector size then each operation is slower than necessary for security. EVA does instruct the user in these cases to consider possible optimizations.

EXL vector above introduces yet another level of vector sizes and, for programs that use EXL vectors exclusively, mostly hides the EVA vector size. However, users must still specify `vec_size` for the `EvaProgram` constructor and different choices for this parameter will result in very different performance. EXL does improve the situation over vanilla EVA by allowing different values for the EVA vector size to be tried with no other changes to the program, but ideally this would be handled transparently by EVA.

We propose remedying this by having users provide an *program generator* instead of a concrete EVA program. The example program in Section 3.3 would for example be expressed as:

```
def prog():
  v = VecInput('v', 5)
```

Require: GeneratePrograM works for al v s.t. $v \geq v_{min}$ and $v_{large} \geq v_{min}$.
Ensure: The selected P, Q, v are secure and P was generated with a $v' \geq v_{min}$.

```
1: procedure IterateUp(v_min)
2:     v ← v_min
3:     loop
4:         P, Q, v_secure ← TryVecSize(v)
5:         if v ≥ v_secure then
6:             return P, Q, v
7:         else
8:             v ← 2 · v
9: procedure TryVecSize(v)
10:    P ← GenerateProgram(v)
11:    P', Q ← Compile(P)
12:    v' ← MinSlotsForModulus(Q)
13:    return P', Q, v'
```

Algorithm 5: Simple vector size selection procedures

```
w = VecInput('w', 5)
VecOutput('y', vec_dot_product(v, w))
```

PyEVA would use this to generate candidate programs with different `vec_size` arguments and select a good one based on some criteria.

Algorithm 5 illustrates a simple way to select an EVA vector size given a user-provided GenerateProgram procedure and a minimum vector size the program needs. IterateUp will find the smallest valid vector size that is secure by iterating up from the minimum. This, however, has the drawback that the program will be compiled potentially many times. We leave designing a good general purpose selection procedure for future work.

3.5 Statistical Functions and More

We have added a set of common statistical functions to EXL, as well as a polynomial approximation generator for evaluating non-polynomial functions. These were implemented using the EXL vector class and the functions for horizontal sum and dot product described in Section 3.3. Thanks to PyEVA's expressiveness, implementing them is simple:

```
import numpy as np

def average(x):
  return vector_horizontal_sum(x) / len(x)

def sum_of_squares(x):
  return vector_horizontal_sum(x**2)

def sum_of_squared_errors(avg, x):
  return sum_of_squares(x - avg)

def variance(x):
  x = sum_of_squared_errors(average(x), x)
  return x / (len(x) - 1)

def poly_approx(fun, low, up, degree):
  def poly(x):
    ...
    return y
  return poly
```

```
# Standard deviation
def sd(x):
  sqrt = poly_approx(np.sqrt, 0, 100, degree=6)
  return sqrt(variance(x))

def correlation(x, y):
  avg_x = average(x)
  avg_y = average(y)
  sum_xy = vector_dot_product(x-avg_x, y-avg_y)
  mul_sd_xy = sd(x) * sd(y) * len(x)
  return (sum_xy, mul_sd_xy)
```

We have omitted the polynomial approximation implementation above, as choice of the method is very application-dependent. We leave the task of designing a comprehensive library of polynomial approximation methods for future work.

The code above serves as a good example of how EXL functions build on both the EXL vector class and each other. This demonstrates the benefit of building a comprehensive library in PyEVA. And we hope to start a virtuous cycle of developers contributing to EXL and making future contributions easier while doing so.

4 Conclusions and Future Work

In this paper we have demonstrated valuable improvements to the EVA toolchain, providing further evidence that the approach EVA is taking towards a CKKS-specific compiler toolchain can provide good performance and a far better user-experience than native CKKS API in any library. We have demonstrated that building higher level libraries on top of EVA is meaningful and can hugely aid in simple use-cases of CKKS, e.g., implementing simple statistical functions, or arbitrary-sized vector computations. Concretely, we built the beginnings of an EVA Extension Library (EXL). As future work, we believe EXL can be augmented with a much richer set of data types and functions, e.g., tensors and various implementations for neural network kernels.

There are multiple direction for extending EVA itself. As was discussed in Section 3.4, the EVA vector size is a tricky concept that would ideally be abstracted away by EXL, which would require yet another layer of abstraction in the program creation.

Another useful feature that EVA does not currently support is combining inputs from multiple data sources. For example, a computation may calculate the correlation between data from two data owners, both holding the same public EVA context (public key). However, EVA requires the user to specify all input wires at once and does not allow one party to specify part of the inputs.

While EVA currently targets only CKKS, we believe there may be benefit in targeting BFV and BGV as well. Both BGV and BFV would require a noise estimator to be built into EVA, as SEAL currently does not include such an estimator. However, some other libraries, notably HElib [27], already implement noise estimators, which EVA could leverage directly. Extending EVA to support other library back-ends, targeting either CKKS or BFV/BGV, would be valuable.

On the usability side several open questions remain. Does EVA and EXL provide a sufficient level of abstraction to make homomorphic encryption, and the CKKS scheme in particular, generally available to developers without extensive training? Cryptographic libraries are notorious for poor usability [26, 32], and homomorphic encryption libraries are undoubtedly extremely challenging to use for non-experts. Performing usability studies would help compiler developer teams identify directions to pursue in the future.

Porcupine [15] takes a interesting approach in attempt to help users vectorize their computations. Since vectorization is essential to making CKKS (also BFV/BGV) applications meaningfully performant, it would make sense to combine a tool such as Porcupine with EVA. Currently EVA/EXL users would need to understand to utilize the EXL vector classes to the extreme to achieve good performance, which is not going to be obvious for normal developers.

References

1. Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Homomorphic encryption security standard. Tech. rep., HomomorphicEncryption.org, Toronto, Canada (November 2018)
2. Archer, D.W., Calderón Trilla, J.M., Dagit, J., Malozemoff, A., Polyakov, Y., Rohloff, K., Ryan, G.: Ramparts: A programmer-friendly system for building homomorphic encryption applications. In: Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 57–68 (2019)
3. Benaissa, A., Retiat, B., Cebere, B., Belfedhal, A.E.: Tenseal: A library for encrypted tensor operations using homomorphic encryption. arXiv preprint arXiv:2104.03152 (2021)
4. Boemer, F., Costache, A., Cammarota, R., Wierzynski, C.: 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. pp. 45–56 (2019)
5. Boemer, F., Lao, Y., Cammarota, R., Wierzynski, C.: ngraph-he: A graph compiler for deep learning on homomorphically encrypted data. In: Proceedings of the 16th ACM Int'l Conf. on Computing Frontiers. pp. 3–13 (2019)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
7. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Annual cryptology Conf. pp. 505–524. Springer (2011)
8. Carpov, S., Dubrulle, P., Sirdey, R.: Armadillo: A compilation chain for privacy preserving applications. In: Proceedings of the 3rd Int'l Workshop on Security in Cloud Computing. pp. 13–19 (2015)
9. Chen, H., Chillotti, I., Song, Y.: Improved bootstrapping for approximate homomorphic encryption. In: Annual Int'l Conf. on the Theory and Applications of Cryptographic Techniques. pp. 34–54. Springer (2019)
10. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: Annual Int'l Conf. on the Theory and Applications of Cryptographic Techniques. pp. 360–384. Springer (2018)
11. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full rns variant of approximate homomorphic encryption. In: Int'l Conf. on Selected Areas in Cryptography. pp. 347–368. Springer (2018)
12. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Int'l Conf. on the Theory and Application of Cryptology and Information Security. pp. 409–437. Springer (2017)
13. Chielle, E., Mazonka, O., Tsoutsos, N.G., Maniatakos, M.: E3: A framework for compiling c++ programs with encrypted operands. IACR Cryptol. ePrint Arch. **2018**, 1013 (2018)
14. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Tfhe: Fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)
15. Cowan, M., Dangwal, D., Alaghi, A., Trippel, C., Lee, V.T., Reagen, B.: Porcupine: A synthesizing compiler for vectorized homomorphic encryption. arXiv preprint arXiv:2101.07841 (2021)
16. Crockett, E., Peikert, C.: λολ: Functional lattice cryptography. In: Proceedings of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. pp. 993–1005 (2016)
17. Crockett, E., Peikert, C., Sharp, C.: Alchemy: A language and compiler for homomorphic encryption made easy. In: Proceedings of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. pp. 1020–1037 (2018)
18. Dathathri, R., Kostova, B., Saarikivi, O., Dai, W., Laine, K., Musuvathi, M.: EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In: Proceedings of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 546–561 (2020)
19. Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., Mytkowicz, T.: Chet: An optimizing compiler for fully-homomorphic neural-network inferencing. In: Proceedings of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 142–156 (2019)
20. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Annual Int'l Conf. on the Theory and Applications of Cryptographic Techniques. pp. 617–640. Springer (2015)
21. van Elsloo, T., Patrini, G., Ivey-Law, H.: Sealion: A framework for neural network inference on encrypted data. arXiv preprint arXiv:1904.12840 (2019)

22. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. **2012**, 144 (2012)
23. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM Symp. on Theory of computing. pp. 169–178 (2009)
24. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Annual Cryptology Conf. pp. 75–92. Springer (2013)
25. Gorantala, S., Springer, R., Purser-Haskell, S., Lam, W., Wilson, R., Ali, A., Astor, E.P., Zukerman, I., Ruth, S., Dibak, C., et al.: A general purpose transpiler for fully homomorphic encryption. arXiv preprint arXiv:2106.07893 (2021)
26. Green, M., Smith, M.: Developers are not the enemy!: The need for usable security apis. IEEE Security & Privacy **14**(5), 40–46 (2016)
27. Halevi, S., Shoup, V.: Algorithms in helib. In: Annual Cryptology Conf. pp. 554–571. Springer (2014)
28. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Cryptographers' Track at the RSA Conf. pp. 364–390. Springer (2020)
29. Lattigo v2.1.1. `http://github.com/ldsec/lattigo` (Dec 2020), ePFL-LDS
30. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Proceedings of the Twenty-Fourth ACM Symp. on Operating Systems Principles. p. 456–471. SOSP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2517349.2522739, `https://doi.org/10.1145/2517349.2522739`
31. PALISADE Lattice Cryptography Library (release 1.11.2). `https://palisade-crypto.org` (May 2021)
32. Patnaik, N., Hallett, J., Rashid, A.: Usability smells: An analysis of developers' struggle with crypto libraries. In: Fifteenth Symp. on Usable Privacy and Security ({SOUPS} 2019) (2019)
33. Rivest, R.L., Adleman, L., Dertouzos, M.L., et al.: On data banks and privacy homomorphisms. Foundations of secure computation **4**(11), 169–180 (1978)
34. Microsoft SEAL (release 3.6). `https://github.com/Microsoft/SEAL` (Nov 2020), microsoft Research, Redmond, WA.
35. Sheep is a homomorphic encryption evaluation platform. `https://github.com/alan-turing-institute/SHEEP` (Nov 2019)
36. University, S.N.: Heaan. `https://github.com/snucrypto/HEAAN` (Dec 2020)
37. Viand, A.: Sok: Fully homomorphic encryption compilers. In: IEEE Symp. on Security and Privacy (2021)
38. Viand, A., Shafagh, H.: Marble: Making fully homomorphic encryption accessible to all. In: Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 49–60 (2018)