Pyfhel: PYthon For Homomorphic Encryption Libraries

Alberto Ibarrondo
ibarrond@eurecom.fr
IDEMIA & EURECOM, France

Alexander Viand
alexander.viand@inf.ethz.ch
ETH Zurich, Switzerland

ABSTRACT

Fully Homomorphic Encryption (FHE) allows private computation over encrypted data, disclosing neither the inputs, intermediate values nor results. Thanks to recent advances, FHE has become feasible for a wide range of applications, resulting in an explosion of interest in the topic and ground-breaking real-world deployments. Given the increasing presence of FHE beyond the core academic community, there is increasing demand for easier access to FHE for wider audiences. Efficient implementations of FHE schemes are mostly written in high-performance languages like C++, posing a high entry barrier to novice users. We need to bring FHE to the (higher-level) languages and ecosystems non-experts are already familiar with, such as Python, the de-facto standard language of data science and machine learning. We achieve this through wrapping existing FHE implementations in Python, providing one-click installation and convenience in addition to a significantly higher-level API. In contrast to other similar works, Pyfhel goes beyond merely exposing the underlying API, adding a carefully designed abstraction layer that feels at home in Python. In this paper, we present Pyfhel, introduce its design and usage, and highlight how its unique support for accessing low-level features through a high-level API makes it an ideal teaching tool for lectures on FHE.

CCS CONCEPTS

• **Security and privacy** → **Usability in security and privacy**; **Public key encryption**.

KEYWORDS

Fully Homomorphic Encryption; FHE; Python; Abstraction

ACM Reference Format:
Alberto Ibarrondo and Alexander Viand. 2021. Pyfhel: PYthon For Homomorphic Encryption Libraries. In *Proceedings of the 9th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '21), November 15, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3474366.3486923

1 INTRODUCTION

Misuse of private data not only harms users, but also threatens the adoption of new technological innovations — e.g., the risks and complexities associated with sharing medical data are widely considered as a barrier to advances in personalized medicine. Today, industry best practices require that service providers protect personal data in-transit and when at rest using encryption. However, this data must be decrypted before being used for computations, requiring the service provider to have access to the keying material. This exposes the data to multiple threats, including abuse by malicious actors. Therefore, demand has increased for solutions that can protect personal data while preserving the utility of services.

Fully Homomorphic Encryption (FHE) allows a third party to perform computations on encrypted data without learning the inputs or the computation results. In contrast to *partially* homomorphic encryption, which supports only one type of arithmetic operation (e.g. only additions), *fully* homomorphic encryption allows encrypted multiplications and additions, theoretically allowing private computation of arbitrary functions. This concept was conceived by Rivest et al. in the 1970s, but it remained unrealized until Craig Gentry presented a first feasible FHE scheme in 2009. Since then, FHE has gone from theoretical breakthrough to practical deployment, dropping the initial 30 minutes required to compute a multiplication between two encrypted values down to less than 20 milliseconds. Even then, FHE multiplications are still around seven orders of magnitude slower than native CPU integer multiplication instructions. Therefore, FHE requires, like other secure computation solutions, that applications be specifically adapted and optimized.

FHE is starting to be deployed in widely used mainstream software. For example, Microsoft's Edge browser uses FHE in its privacy-preserving password monitor [8, 9, 23], which compares users' login information to known leaks without revealing users' sensitive information to the service. In the medical domain, there has been significant work on using FHE to enable large-scale genome-wide association studies (GWAS) [3, 25]. In the domain of machine learning, FHE has been applied to train logistic regression [6, 24] models and to run privacy-preserving inference for neural networks [7, 14, 15, 20]. FHE-based secure computation solutions have generated significant commercial interest and Gartner projects [16] that "by 2025, at least 20% of companies will have a budget for projects that include fully homomorphic encryption."

Given the increasing presence of FHE beyond the core academic community, we must provide easy access to FHE for a wider audience. While there exists a variety of high-quality open-source implementations of different modern FHE schemes [12, 13, 21, 27, 30], these are mostly written in C++ or other high-performance systems languages, in a common approach for cryptographic code seeking maximal performance for heavy-weight operations. However, languages like C++ are significantly less popular [31], especially outside of the core computer science community, than higher-level languages like Python, which has established itself as the de-facto standard language of data science and especially machine learning.

In the interest of promoting FHE among less technical users, we need to develop *wrappers*, which expose interfaces for the underlying cryptographic libraries in different languages, e.g., Python.

WAHC '21, November 15, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8656-2/21/11.
https://doi.org/10.1145/3474366.3486923

Beyond merely providing a way to access, e.g., functionalities from a C++ library in a different language, well-written wrappers should provide idiomatic ways to use the code, respecting best practices and conventions of the target language. This results in code that feels familiar to developers and allows them to correctly and efficiently use the library. Additionally, these wrappers should ideally abstract away the sometimes complex installation process of these libraries. Most existing FHE implementations require, or strongly benefit from, dependencies which need to be installed through properly configured toolchains to make the library work properly. Wrappers that handle this setup and provide an automatic one-click installation greatly improve the practical accessibility of FHE.

Contributions. With Pyfhel, we provide a Python wrapper for the Microsoft SEAL [30] library, extendable to other C++ libraries, that goes beyond merely exposing the underlying API by adding a carefully designed abstraction layer that feels at home in Python. Pyfhel offers *(i)* one-click installation, including the underlying libraries, *(ii)* a high-level Python-first abstraction layer that makes working with FHE significantly easier, including *(iii)* high-level APIs for low-level functionalities not generally exposed. We show how Pyfhel can not only assist developers in exploring FHE, but also how it is specifically well suited to use in FHE education.

Related Work. There already exists a plethora of Python wrappers for FHE libraries, many of them unmaintained and outdated. Most rely on automatic C++ wrapping tools like `pybind11` [2, 32] or `Boost.Python` [27], which requires large parts of the wrapper logic to be written in C++ to preserve performance. PySEAL [32] is such a no-longer-maintained `pybind11` wrapper. Many require the user to compile the underlying library themselves, using a Unix-only toolchain, like the more recent SEAL-Python [22]. TenSEAL [2], which appeared several years after the initial release of Pyfhel, shows the most promise. It is `pybind11`-based and features a one-click setup, but focuses mostly on high level Machine Learning and tensor operations. Other approaches (e.g., pyFHE [17]) implement schemes directly in Python, at the cost of significantly slower operations. Finally, FHE libraries have also experimented with Python interfaces, including PALISADE[27], and the EVA compiler [15] for SEAL. However, both still require non-Python build toolchains. While TenSEAL and EVA are great for novice users, they do not offer proper access to the underlying data structures, which is required to, e.g., understand ciphertext maintenance in educational settings. We argue that, just as a healthy FHE ecosystem requires different libraries implementing the same schemes, it also benefits from different ways to expose this functionality.

2 PRELIMINARIES

A *homomorphic* encryption scheme allows computation over the ciphertexts that results in ciphertexts encrypting the result of the equivalent plaintext operation. An *additively* homomorphic scheme holds $\mathrm{Dec}(\mathrm{Enc}(a) \oplus \mathrm{Enc}(b)) = \mathrm{Dec}(\mathrm{Enc}(a+b))$, where $+$ is the addition operation over plaintexts, and $\oplus$ is an operation over the ciphertexts. Similarly, a *multiplicatively* homomorphic encryption scheme supports $\mathrm{Dec}(\mathrm{Enc}(a) \otimes \mathrm{Enc}(b)) = \mathrm{Dec}(\mathrm{Enc}(a \times b))$ where $\times$ is the multiplication operation over the plaintexts and $\otimes$ is an operation over the ciphertexts. *Fully* homomorphic encryption, i.e., encryption that is both additively and multiplicatively homomorphic, was conceptually proposed equally early [29] but the first feasible FHE scheme was developed only in 2009 [19]. Because multiplication and addition can emulate `AND` and `OR` gates, respectively, over binary plaintexts, fully homomorphic encryption allows arbitrary computations to be performed.

2.1 FHE Schemes

Virtually all modern FHE schemes are based on (variants of) the Learning with Errors (LWE) hardness assumption [28] and rely on a small amount of *noise* added during encryption to guarantee security. During homomorphic operations, this noise grows. This effect is negligible for additions, but very significant for multiplications. Should the noise grow too large, decryption would no longer produce correct results. Theoretically, a technique known as *bootstrapping* can be used to homomorphically reset the noise in a ciphertext. However, this can be computationally expensive and therefore is not frequently used in practice. Instead, schemes are instantiated with parameters large enough to allow the computation to complete without requiring bootstrapping. In the following, we briefly introduce the Brakerski/Fan-Vercauteren (BFV) [4, 18] and Cheon-Kim-Kim-Song (CKKS) [11] schemes implemented in the SEAL library [30]. We omit formal definitions, referring the reader to the original papers for more detail.

BFV. The Brakerski/Fan-Vercauteren scheme [4, 18] features powerful Single Instruction Multiple Data (SIMD) parallelism, making it highly efficient for applications working over larger amounts of data. Messages are vectors of integers $\vec{m} \in \mathbb{Z}^n$ which are encoded into plaintext polynomials of degree n. Simply encoding each vector element into a coefficient would lead to issues during homomorphic multiplications, and instead the vectors are mapped to polynomials using the Chinese Remainder Theorem (CRT). This process is generally transparent to the user, who can simply treat ciphertexts as encrypted vectors of integers. Homomorphic addition and multiplication operate element-wise, giving rise to the SIMD parallelism. Additionally, *rotation* operations cyclically rotate the elements inside the vectors, allowing elements originally stored at different indices (also known as "slots") to interact.

While BFV supports bootstrapping in theory, it is slow and thus not commonly used. Instead, the scheme is generally instantiated with parameters large enough to handle the noise growth. Since multiplications incur in high noise growth, its number (the *multiplicative depth*) should be as small as possible. Even then, an operation known as *relinearizaton*, which reshapes the ciphertext without changing the underlying message, should be used between multiplications to help limit noise growth and ciphertext size.

CKKS. The Cheon-Kim-Kim-Song scheme [11] implements *approximate* homomorphic encryption, i.e., $\mathrm{Dec}(\mathrm{Enc}(m)) \approx m$. In traditional FHE schemes, should the noise grow large enough to affect the message, this is considered an invalid ciphertext and decryption fails. In CKKS, in contrast, noise that ends up overlapping the least significant bits of a message is considered to be part of the message, leading to the approximate nature of the scheme.

CKKS is designed to be used with vectors of messages $\vec{m} \in \mathbb{R}^n$, i.e., fractional values, and encoding applies a *scaling factor*, i.e. computes $\lfloor m * \Delta \rceil$, where Δ is a large integer (e.g., 2^{30}). While this type of encoding can be used in other schemes, too, this quickly

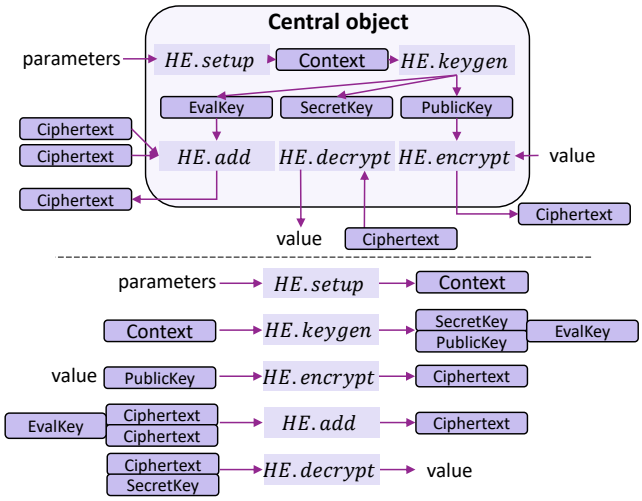

Figure 1: Example of centralized (top, ours) vs functional design (bottom) approaches in FHE scheme APIs

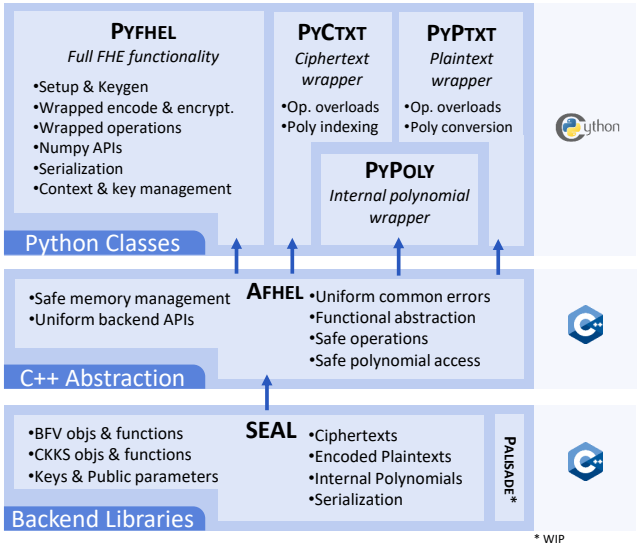

Figure 2: Pyfhel high level layered architecture

leads to issue with subsequent multiplications, as the scale will grow to Δ^2, Δ^3, etc. with each multiplication. Since Δ is large, this will quickly lead to the encoded message being reduced modulo q, producing incorrect results. CKKS introduces a technique to combat this growth, introducing the ability to *rescale* a ciphertext, essentially homomorphically dividing it by Δ.

While the underlying design of CKKS is closer to the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [5] than to BFV, using CKKS is very similar from a developer perspective: homomorphic operations offer SIMD parallelism, rotations are available, and relinearization should be used after ciphertext-ciphertext multiplications.

3 DESIGN

3.1 Design principles

The design of Pyfhel adheres to several key principles. In terms of programming language we rely on `C++` to preserve the high efficiency of backend libraries, and Cython [1] (a superset of `C/C++` and Python) to bridge the gap with Python, fusing `C/C++` performance with Python-like expressiveness and dynamic typing. Pyfhel features a *one-click setup* that automatically installs all backends with the library, requiring no knowledge on compilation toolchains:

```
pip install Pyfhel      # Backends inside!
```

At a high level we opt for a *centralized* approach (Figure 1 (top)), where a single central class holds most of the functionalities and keeps track of the objects that rarely change after setup, including contexts and keys. Whereas the functional approach (Figure 1 (bottom)) common in FHE implementations requires the user to manually understand and keep track of the appropriate context and order of API calls, the centralized design style of Pyfhel understands and watches the global state of the FHE scheme and can raise informative errors to guide users towards the proper way to use the library, or even infer the missing pieces (e.g., generating a rotation key if not present when performing rotation).

Pyfhel is also designed to cleanly expose the low-level polynomials that make up keys and plaintexts/ciphertexts. While possible in the underlying libraries, this is generally not part of the "porcelain" API intended for developers, but rather low-level "plumbing" that is documented sparingly (if at all). For example, SEAL uses a combination of custom iterator and pointer classes to deal with the underlying polynomials, manually tracking the number and sizes of the individual data elements. Pyfhel instead abstracts it away via a high-level interface similar to that used for plain- and ciphertexts.

3.2 Architecture

In order to realize these design principles, Pyfhel uses a layered architecture consisting of three key layers, as seen in Figure 2.

(1) *Backend libraries*: The unmodified, up-to-date FHE libraries generally written in C++, automatically loaded from their official sources. These expose homomorphic operations, keys and context parameters, ciphertexts & encoded plaintexts, and serialization features. Using these correctly requires managing significant amounts of state information in the application code. Pyfhel currently supports SEAL [30], with work on PALISADE [27] under way.

(2) *Afhel*: Our **A**bstraction **f**or **H**omomorphic **E**ncryption **L**ibs acts as a safe and uniform `C++` encapsulation of different backend APIs. In addition, it manages memory, offering memory-safe versions of low-level APIs, and tracks the state required to interact with the libraries properly. The encapsulation and abstraction offered by Afhel is key in enabling us to build clean and easy-to-use interfaces.

(3) *Python Classes*: Pyfhel's python classes expose all FHE functionalities in a pythonic way, allowing users to work in a familiar setting, writing code that often looks like pseudocode. The `Pyfhel` class centralizes a variety of functionalities, including state management, tracking keys and context required for operations on ciphertexts/plaintexts. The `PyCtxt` and `PyPtxt` classes wrap ciphertexts and plaintexts respectively, allowing users to express arithmetic expressions simply using operator overloads. These classes also offer access

to the underying polynomials via simple indexing. The polynomials are wrapped by the `PyPoly` class which offers similar arithmetic operators and allows seamless conversion to and from arrays/lists of coefficients.

4 USING PYFHEL

In this section, we demonstrate how to use Pyfhel through a series of examples. Extended versions and additional examples documenting further features are available in the Pyfhel repository [1].

4.1 Setup and Parameters

All computations begin by creating a Pyfhel object, initializing a scheme with chosen parameters and generating/loading keys.

```
from Pyfhel import PyCtxt, Pyfhel, PyPtxt
HE = Pyfhel()
HE.contextGen(scheme='BFV', n=16384, p=65537)
HE.keyGen()
```

In terms of parameters, n (polynomial degree) determines the number of slots the plaintext vectors have (n in BFV and $n/2$ in CKKS). Meanwhile, p (plaintext modulus) determines the modulus of the plaintext space in BFV, which determines how large encrypted values can get before wrap-around occurs (e.g., $65537 = 2^{16}$ which is equivalent to working with 16-bit unsigned integers). One could also provide the ciphertext modulus q, which determines how much noise can accumulate before decryption fails. Larger q can tolerate more noise, and therefore more complex computations, but also lead to slower homomorphic operations and weaker security if n stays fixed. For *BFV*, q is set using the largest value that still achieves 128/192/256-bit security for the given polynomial degree (parameter *sec*). Instead of providing q, users working with CKKS must provide a modulus chain of prime sizes q_i (e.g., `qs=[30,30,30,30,30]`). Although not usually required, it is possible to define other setup parameters, as described in the Pyfhel documentation, for expert users seeking control over lower-level aspects.

4.2 Encryption & Decryption

In order to encrypt messages, the values must first be *encoded* into plaintext objects (`PyPtxt`). Similar, after decryption, the resulting plaintext must be *decoded*. Both BFV and CKKS internally feature vector-like plaintext spaces. Pyfhel is able to encode a variety of different datatypes, including lists/numpy arrays and single values, where Pyfhel repeats the value until all slots are filled.

```
integer = 45
int_ptxt = HE.encode(integer)   # PyPtxt [45, 45,...]
int_ctxt = HE.encrypt(int_ptxt) # PyCtxt

import numpy as np
np_array = np.array([6, 5, 4, 3, 2, 1],dtype=np.int64)
array_ptxt = HE.encode(np_array)    # Accepts list too
array_ctxt = HE.encrypt(array_ptxt) # PyCtxt

# Decrypt and Decode
ptxt_dec = HE.decrypt(int_ctxt)   # PyPtxt
integer_dec = HE.decode(ptxt_dec) # 45

# One-step encryption/decryption
array_ctxt = HE.encrypt(np_array)
array_dec  = HE.decrypt(array_ctxt, decode=True)
```

[1] https://github.com/ibarrond/Pyfhel/tree/master/examples

4.3 Homomorphic Operations

The core operations of a homomorphic scheme are the addition and multiplication operation. However, there are a variety of other operations, including ciphertext maintenance operations like relinearization and rescaling. In addition to ciphertext-ciphertext operations, FHE schmes also offer ciphertext-plaintext operations that are faster and lead to noise growth. Pyfhel provides operator overloads for arithmetic operations ($+, -, *$ and $+=, -=, *=$ for in-place operations), which automatically select the appropriate type of operation depending on the operands. One can also use values directly in computations, with Pyfhel automatically encoding them into suitable plaintext objects.

```
ptxt_a = HE.encode(-12)
ptxt_b = HE.encode(34)
ctxt_a = HE.encrypt(56)
ctxt_b = HE.encrypt(-78)

# ctxt-ctxt operations
ctxt_s = ctxt_a + ctxt_b # or ctxt_a += ctxt_b (in place)
ctxt_m = ctxt_a * ctxt_b

# ctxt-ptxt operations
ctxt_s_p = ptxt_a + ctxt_b # or 12 + ctxt_b
ctxt_m_p = ctxt_a * ptxt_b # or ctxt_a * 34

# maintenance operations
HE.relinKeyGen()
HE.relinearize(ctxt_s)  # requires relinKey

# rotations (length n)
ctxt_c = HE.encrypt([1,2,3,4])
ctxt_rotated = ctxt_c << 1 # [2,3,4,0,...,0,1]
```

4.4 IO & Serialization

Pyfhel has full support for serialization, which is not only useful to store generated keys but can also be used to realize true client-server computations. In the following example, we create two independent Pyfhel instances, one representing the client and one representing the server. Only the client-object has access to the secret keys and can decrypt messages. For simplicity, we simulate communication using the file system, but this could easily be exchanged for a real communication channel.

```
##### CLIENT
HE = Pyfhel()
HE.contextGen(scheme='BFV', n=4096, p=65537)
HE.keyGen() # Generates public and private key
# Save context and public key only
HE.savepublicKey("mypk.pk")
HE.saveContext("mycontext.con")
# Encrypt and save inputs
ctxt_a = HE.encrypt(15) # implicit encoding
ctxt_b = HE.encrypt(25)
ctxt_a.to_file("ctxt_a.ctxt")
ctxt_b.to_file("ctxt_b.ctxt")

##### SERVER
HE_server = Pyfhel()
HE_server.restoreContext("mycontext.con")
HE_server.restorepublicKey("mypk.pk") # no secret key
# Load ciphertexts
```

```
ca = PyCtxt(pyfhel=HE_server, fileName="ctxt_a.ctxt")
cb = PyCtxt(pyfhel=HE_server, fileName="ctxt_b.ctxt")
# Compute homomorphically and send result
cr = (ca + cb) * 2
cr.to_file("cr.ctxt")

##### CLIENT
# Load and decrypt result
c_res = PyCtxt(pyfhel=HE, fileName="cr.ctxt")
print(c_res.decrypt())
```

5 USING PYFHEL IN EDUCATION

Pyfhel can be used as an excellent tool for integrating FHE into teaching. Due to its one-click-install, it is much more feasible to incorporate Pyfhel coursework into a curriculum than with the underlying C++ libraries. By providing abstractions, syntactic sugar (e.g., operator overloads) and other conveniences, Pyfhel is considerably more concise and allows students to focus on the task at hand. Beyond providing ease and accessibility, Pyfhel includes access to low-level features to enable teaching. Specifically, Pyfhel allows users to easily access the underlying polynomials that actually make up plaintexts and ciphertexts. Working with the underlying polynomials is not generally necessary to employ FHE, but it can be helpful in teaching situations to be able to dissect ciphertexts and study elements individually. In the following, we present two case studies for using Pyfhel in teaching. One focuses on the challenges of managing scales in CKKS, while the other uses the polynomial API in Pyfhel to study a key recovery attack on CKKS.

5.1 Exploring common CKKS pitfalls

Implementing applications in FHE can be challenging for novice users, e.g.,due to failing to properly manage scaling factors throughout a CKKS-based computation. While higher-level tools and compilers like EVA [15] increasingly provide automated solutions for these challenges, exploring them in a teaching setting continues to have value, since it requires transferring theoretical information about the scheme to practical application.

We show how to use Pyfhel to explore several pitfalls in working with CKKS, using the computation $((x + y) * (z * 5)) + 10$, where x, y and z are (secret) inputs. First, we set up context and keys:

```
from Pyfhel import PyCtxt, Pyfhel, PyPtxt
HE = Pyfhel()
HE.contextGen(scheme='CKKS', n=16384,qs=[30,30,30,30,30])
HE.keyGen()
```

Now we can perform the ciphertext-ciphertext addition $x + y$ and the ciphertext-plaintext multiplication $z * 5$ before performing the ciphertext-ciphertext multiplication between those two results.

```
ctxt_x = HE.encrypt(3.1, scale=2 ** 30) # implicit encode
ctxt_y = HE.encrypt(4.1, scale=2 ** 30)
ctxt_z = HE.encrypt(5.9, scale=2 ** 30)

ctxtSum = ctxt_x + ctxt_y
ctxtProd = ctxt_z * 5
ctxt_t = ctxtSum * ctxtProd
```

Next, we explicitly encode the constant 10 the same as we encoded the inputs x, y and z. This will lead to an error since the scale of `ctxtProd` has increased to 2^{60} after the first multiplication, and multiplying it with `ctxtSum` which is still at scale 2^{30} (addition does not change scale) causes `ctxt_t` to have scale 2^{90}. Since, in fixed-point arithmetic, addition can only performed over numbers represented at the same scale, the addition will fail.

```
ptxt_ten = HE.encode(10, scale=2 ** 30)
ctxt_result = ctxt_t + ptxt_ten #error: mismatched scales
```

Of course, this can be resolved by encoding 10 at the correct scale, i.e., setting `ptxt_ten=HE.encode(10, scale=2**90)`. Alternatively, we can use `ctxt_result=ctxt_t+10` and Pyfhel will automatically deduce the correct scale. However, if instead of a constant we have an input d that the user encrypted at the same scale as all other inputs, this solution no longer applies. Instead, we must use *rescaling* to homomorphically decrease the scale of `ctxt_t` to the initial scale. In CKKS, each rescaling reduces the scale down by one "step" (here $\Delta = 2^{30}$), so we need to perform two consecutive rescaling operations.

```
ptxt_d = HE.encode(10, 2 ** 30)
ctxt_d = HE.encrypt(ptxt_d)
HE.rescale_to_next(ctxt_t)  # 2^90 -> 2^60
HE.rescale_to_next(ctxt_t)  # 2^60 -> 2^30
```

Surprisingly, multiplying `ctxt_t` and `ctxt_d` will still fail, due to how efficient versions of CKKS implement rescaling [10]. In essence, rescaling also decreases the ciphertext modulus, and we need to decrease the ciphertext modulus of `ctxt_d` to match.

```
HE.mod_switch_to_next(ctxt_d) # match first rescale
HE.mod_switch_to_next(ctxt_d) # match second rescale
```

Now, trying to compute `ctxt_t+ctxt_d` will no longer produce an error about mismatched moduli. However, even though we rescaled, it still gives an error about mismatched scales. Due to subtleties in the the way rescaling works, instead of dividing the scale by exactly the step size (2^{30}), it divides the scale by a prime number very close, but not exactly equal to, the step size. Therefore, `ctxt_t` is now actually at a scale of, e.g., $2^{29.86\cdots}$. Since CKKS is inherently approximate to begin with, we can ignore this difference and simply accumulate it into the overall approximation error. In order to do this, we manually override the scaling factor used, which finally allows the addition to complete successfully.

```
ctxt_t.set_scale(2**30)
ctxt_result = ctxt_t + ctxt_d # final result
```

5.2 Implementing Key-Recovery for CKKS

Allowing low-level access to polynomials enables implementing a variety of advanced techniques or attacks, including the CKKS key recovery attack by Li and Micciancio [26]. Their key insight is that *noisy* decryption reveals information about the secret key. In a non-approximate scheme, knowing the input x and the function f allows one to perfectly simulate the homomorphic computation and derive $f(x)$. However, in an approximate scheme like CKKS, the decryption will be $y \approx f(x)$ and, importantly, the differences between y and $f(x)$ depend on the secret key. Interestingly, this attack does not contradict the security guarantees proven for CKKS, as the attack is outside the IND-CPA model. We briefly describe the simplest form of the attack below.

In CKKS, a ciphertext ct has two components, i.e., $ct = (a, b)$ where $b = a * s + m + e$, for secret key s, random mask a, and noise term e. The decryption of ct is $c := \mathrm{Dec}_s(ct) = b - a * s = m + e$. Here, $m = f(x)$, which we assume is known to the adversary. If the adversary also gains access to the decryption c, they can solve

the linear equation $a * s = b - c$ by computing the multiplicative inverse a^{-1} (which exists with high probability), recovering the secret key. Note that this ignores the encoding and decoding used in CKKS. Instead of seeing the plaintext, it is more realistic to assume the attacker only has access to the decoded message. While the encoding is not actually perfectly reversible, simply re-encoding the decoded value is frequently sufficient to enable the attack.

As we can see below, implementing this attack takes only a few lines of code, using Pyfhel's support for working directly with the underlying polynomials. For comparison, an equivalent C++ implementation of this example, targeting SEAL directly, uses over a hundred lines of code and makes calls to a variety of undocumented low-level features inside SEAL.

```
# Setup: Encrypt, Decrypt, Decode
ctxt = HE.encrypt(0, scale=2**40)
ptxt_dec = HE.decrypt(ctxt)
values = HE.decodeComplex(ptxt_dec)

# Attack
ptxt_re = HE.encode(values, scale=2**40)
a = HE.poly_from_ciphertext(ctxt,1) # PyPoly
b = HE.poly_from_ciphertext(ctxt,0) # or b = ctxt[0]
m = HE.poly_from_plaintext(ptxt_re) # PyPoly
s = (m - b) * ~a # ~a = inverse of a
```

6 DISCUSSION & FUTURE WORK

We have presented Pyfhel, explored its design and usage, including how it can be used as a teaching tool. By providing a python-native abstraction layer on top of existing FHE implementations, Pyfhel makes working with FHE accessible to a significantly wider audience. However, even experts can benefit from the convenience offered by Pyfhel, eliminating potential error sources and reducing time to solution. In the future, we hope to expand the set of supported libraries and to continue our work of creating easy-to-use high-level APIs for low-level features.

REFERENCES

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science Engineering* 13, 2 (2011), 31 –39. https://doi.org/10.1109/MCSE.2010.118

[2] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. 2021. TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption. arXiv:2104.03152 [cs.CR]

[3] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, and Shafi Goldwasser. 2020. Secure large-scale genome-wide association studies using homomorphic encryption. *Proceedings of the National Academy of Sciences of the United States of America* 117, 21 (26 May 2020), 11608–11613. https://doi.org/10.1073/pnas.1918257117

[4] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*. Springer Berlin Heidelberg, 868–886. https://doi.org/10.1007/978-3-642-32009-5_50

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science* (Cambridge, Massachusetts) *(ITCS '12)*. ACM, New York, NY, USA, 309–325. https://doi.org/10.1145/2090236.2090262

[6] Sergiu Carpov, Nicolas Gama, Mariya Georgieva, and Juan Ramon Troncoso-Pastoriza. 2020. Privacy-preserving semi-parallel logistic regression training with fully homomorphic encryption. *BMC medical genomics* 13, Suppl 7 (21 July 2020), 88. https://doi.org/10.1186/s12920-020-0723-0

[7] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. 2017. Privacy-Preserving Classification on Deep Neural Network. *IACR Cryptol. ePrint Arch.* 2017 (2017), 35.

[8] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from fully homomorphic encryption with malicious security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto Canada). ACM, New York, NY, USA. https://doi.org/10.1145/3243734.3243836

[9] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1243–1255. https://doi.org/10.1145/3133956.3134061

[10] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A Full RNS Variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2018*. Springer International Publishing, 347–368. https://doi.org/10.1007/978-3-030-10970-7_16

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*. Springer International Publishing, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology. The Journal of the International Association for Cryptologic Research* 33, 1 (1 Jan. 2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[13] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2020. CONCRETE: Concrete operates oN ciphertexts rapidly by extending TfhE. In *WAHC 2020 – 8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. https://homomorphicencryption.org/wp-content/uploads/2020/12/wahc20_demo_damien.pdf

[14] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 2021. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. Cryptology ePrint Archive, Report 2021/091. https://eprint.iacr.org/2021/091

[15] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal Musuvathi. 2019. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. http://arxiv.org/abs/1912.11951

[16] Mark Driver. 2020. *Emerging Technologies: Homomorphic Encryption for Data Sharing With Privacy*. Technical Report. Gartner, Inc.

[17] Saroja Erabelli. 2020. *pyFHE-a Python library for fully homomorphic encryption*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[18] J Fan and F Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* (2012). https://eprint.iacr.org/2012/144

[19] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph. D. Dissertation. Stanford University. https://crypto.stanford.edu/craig

[20] Laurent Gomez, Alberto Ibarrondo, Marcus Wilhelm, José Márquez, and Patrick Duverger. 2018. Security for Distributed Machine Learning Based Software. In *International Conference on E-Business and Telecommunications*. 111–134.

[21] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481. https://eprint.iacr.org/2020/1481

[22] Huelse. 2020. SEAL-Python. https://github.com/Huelse/SEAL-Python

[23] Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. 2021. Password Monitor: Safeguarding passwords in Microsoft Edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/ Accessed: 2021-7-5.

[24] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. 2018. Logistic regression model training based on the approximate homomorphic encryption. 11-s4 (11 Oct. 2018), 83. https://doi.org/10.1186/s12920-018-0401-7

[25] Miran Kim, Arif Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, Seungwan Hong, Jean-Pierre Hubaux, Duhyeong Kim, Kristin Lauter, Yiping Ma, Lucila Ohno-Machado, Heidi Sofia, Yongha Son, Yongsoo Song, Juan Troncoso-Pastoriza, and Xiaoqian Jiang. 2020. Ultra-Fast Homomorphic Encryption Models enable Secure Outsourcing of Genotype Imputation. *bioRxiv* (2020). https://doi.org/10.1101/2020.07.02.183459

[26] Baiyu Li and Daniele Micciancio. 2021. On the Security of Homomorphic Encryption on Approximate Numbers. In *Advances in Cryptology – EUROCRYPT 2021*. Springer International, 648–677. https://doi.org/10.1007/978-3-030-77870-5_23

[27] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. 2017. *PALISADE lattice cryptography library user manual*. Technical Report. NJIT. https://git.njit.edu/palisade/PALISADE/wikis/resources/palisade_manual.pdf

[28] Oded Regev. 2005. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *In STOC*. ACM Press, 84–93.

[29] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. 1978. On Data Banks and Privacy Homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180. https://people.csail.mit.edu/rivest/RivestAdlemanDertouzos-OnDataBanksAndPrivacyHomomorphisms.pdf

[30] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[31] Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020 Accessed: 2021-7-5.

[32] Alexander J. Titus, Shashwat Kishore, Todd Stavish, Stephanie M. Rogers, and Karl Ni. 2018. PySEAL: A Python wrapper implementation of the SEAL homomorphic encryption library. arXiv:1803.01891 [q-bio.QM]