Manual for Using Homomorphic Encryption for Bioinformatics

Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine,
Kristin Lauter, Michael Naehrig, John Wernsing

Abstract—Biological Data Science is an emerging field facing multiple challenges for hosting, sharing, computing on, and interacting with large data sets. Privacy regulations and concerns about the risks of leaking sensitive personal health and genomic data add another layer of complexity to the problem. Recent advances in cryptography over the last 5 years have yielded a tool, *homomorphic encryption*, which can be used to encrypt data in such a way that storage can be outsourced to an untrusted cloud, and the data can be computed on in a meaningful way in encrypted form, without access to decryption keys. This paper introduces homomorphic encryption to the bioinformatics community, and presents an informal "manual" for using the *Simple Encrypted Arithmetic Library (SEAL)*, which we have made publicly available for bioinformatic, genomic, and other research purposes.

Keywords: homomorphic encryption, outsourced computation, privacy, genome-wide association studies, sequence matching

I. Introduction

A wealth of personal genomic data is becoming available thanks to scientific advances in sequencing the human genome and gene assembly techniques. Hospitals, research institutes, clinics, and companies handling human genomic material and other sensitive health data are all faced with the common problem of securely storing, and interacting, with large amounts of data. Commercial clouds offer solutions but are subject to subpoenas, data misuse or theft, and possible insider attacks. To mitigate the privacy risks inherent in storing and computing on sensitive data, cryptography offers a potential solution in the form of encryption, which metaphorically locks the data in a "box" which requires a key to open. Traditional encryption systems lock data down in a way which makes it impossible to use, or compute on, in encrypted form. Recent advances in cryptography have yielded new tools that allow operations on encrypted data. One such tool is *homomorphic encryption*. Encrypting data using a homomorphic encryption scheme allows for meaningful computation on the encrypted data producing the results of the computation in encrypted form, without the need for decrypting it or requiring access to the decryption key.

This paper details a state-of-the-art homomorphic encryption solution and is meant to serve as a guide to using it for bioinformatics and genomic computations. The Microsoft Research *Simple Encrypted Arithmetic Library (SEAL)* has been publicly released and can be downloaded for experimentation and research purposes[1]. Homomorphic encryption is a technique to encrypt data in such a way that it can be computed on by anyone, without access to the encryption or decryption keys, and the result of the computation is obtained in encrypted form. Solutions for homomorphic encryption which allow one operation, such as addition, have been known for decades, for example based on the RSA or Elgamal cryptosystems. But a homomorphic encryption solution which allows an unlimited number of two operations, i.e. addition and multiplication, enables the computation of *any* circuit, and thus such a solution is referred to as *fully* homomorphic (FHE). The first FHE solution was proposed in [14], and many improvements and extensions have followed over the last 5 years: [9], [6], [15], [30], [1]. For practical applications, an important idea introduced in [25] is to use homomorphic encryption schemes which only allow for a fixed amount of computation on the data. It is usually the case that the computation (the function or the algorithm) which will be applied to the data is known in advance, so that a homomorphic encryption scheme may be instantiated to allow only for that amount of computation on the data. This insight leads to improved parameters, and thus better efficiency for both storage and computation. A scheme for which it is possible, for any fixed given function, to choose parameters such that the scheme allows homomorphic evaluation of that function, is called a *leveled* homomorphic encryption scheme. Using a leveled homomorphic scheme, where parameters are set to allow a certain predetermined, fixed amount of computation, together with application-specific data

[1] The SEAL library is available at http://sealcrypto.codeplex.com/

encodings and algorithmic optimizations, leads to a significant efficiency gain. We refer to such a combination as *Practical* Homomorphic Encryption (PHE).

While it was considered a major breakthrough to develop solutions for homomorphic encryption, serious challenges remain to convert these proposals into practical systems which can overcome the performance hurdles and storage requirements. Significant improvements have been made by encoding data for computation in clever ways to reduce both the size of ciphertexts and the depth of circuits to be evaluated. In Section III we present new methods for encoding real data which lead to concrete improvements in both performance and storage requirements.

When using PHE, parameters of the encoding an encryption schemes should be chosen to optimize for efficiency, while preserving security and correctness. To make this feasible in practice, we have implemented tools (most importantly a *noise growth simulator* and an *automatic parameter selection module*) to help users choose their parameters for maximal performance. Section IV details these techniques, and demonstrates their use in practice.

Homomorphic encryption provides a suitable solution for some, but not all, privacy problems and scenarios. Current solutions allow for a single data owner, such as a hospital, to encrypt data so that it can be securely stored in a commercial cloud. Both private and public key solutions are practical, and with the public key version many parties can upload data encrypted under the hospital's public key: doctors, patients, lab technicians, etc., and the hospital administration can set a policy for access to computations and decryptions. The same idea can be applied to a research institute which stores data in the cloud, selectively allows researchers to make queries on the encrypted data, and then provides decryptions of the results. Consumer and patient scenarios enabled by homomorphic encryption include secure and private outsourcing of storage of personal health records, or of predictive services for risk of disease.

This article is intended to serve as a guide to help bioinformaticians use the library, to experiment with secure computation on biomedical data, and to evaluate the security implications.

II. Homomorphic encryption

An encryption scheme that enables arbitrary computations on encrypted data without first decrypting the data, and without any access to the secret decryption key, is called a Fully Homomorphic Encryption (FHE) scheme. The result of any such computation on encrypted data is itself provided in encrypted form, and can only be decrypted by an owner of the secret key. The first FHE scheme was presented by Gentry [14] in 2009, and since its discovery many improvements and new constructions have been proposed [33], [9], [8], [6], [12], [22], [4], [1], [16], [10].

In Gentry's work, and in many later papers, data is encrypted bit-wise. This means that a separate ciphertext is produced for each bit in the message. The computation is described as a boolean circuit with XOR and AND gates, which can be realized as addition and multiplication modulo 2. Both operations can then be performed on the encrypted bits. Unfortunately, breaking down a computation into bit operations can quickly lead to a large and complex circuit, which together with a significant overhead in memory use per bit imposed by the encryption scheme this makes homomorphic computations very costly.

Fortunately, most known constructions allow for a larger message space. In practical applications the desired computations often only consist of additions and multiplications of integers or real numbers, so there is no need to express the data in a bit-wise manner. Indeed, most known constructions allow to encrypt integers, or appropriate *encodings* of integers, and to homomorphically add and multiply these values. This approach has the clear advantage that a ciphertext now contains much more information than just a single bit of data, making the homomorphic computations significantly more efficient.

In the known FHE schemes, typically ciphertexts contain a certain amount of *inherent noise*, which "pollutes" them. This noise grows during homomorphic operations, and if it becomes too large the ciphertext cannot be decrypted even with the correct decryption key. In order to perform an unlimited number of operations, and thus achieve fully homomorphic encryption, ciphertexts need to be constantly refreshed in order to reduce their noise. This is done using a costly procedure called *bootstrapping* [14].

However, in applications where only a predetermined computation needs to be done, the costly bootstrapping procedure can be avoided by using a so-called leveled

homomorphic encryption scheme. As a guiding principle, the choice of the parameters dictates how many sequential multiplications the computation can involve, i.e. the maximum allowed depth of the computation expressed as an arithmetic circuit, although in reality also other features of the computation need to be taken into account. This approach is often significantly more practical than using an FHE scheme with bootstrapping, but is not as flexible if at a later point a different, more complex function needs to be evaluated.

The remainder of this section describes the leveled homomorphic encryption scheme that is implemented in our Simple Encrypted Arithmetic Library (SEAL).

A. Homomorphic encryption scheme algorithms

The encryption scheme we use is a public-key, homomorphic encryption scheme, and consists of the following algorithms:

- A key generation algorithm KeyGen(parms) that, on input the system parameters parms, generates a public/private key pair (pk,sk) and a public evaluation key evk, which is used during homomorphic multiplication.
- An encryption algorithm Enc(pk,m), that encrypts a plaintext m using the public key pk.
- A decryption algorithm Dec(sk,c), that decrypts a ciphertext c with the private key sk.
- A homomorphic addition operation Add(c_1,c_2) that, given as input encryptions c_1 and c_2 of m_1 and m_2, outputs a ciphertext encrypting the sum $m_1 + m_2$.
- A homomorphic multiplication operation Mult(c_1,c_2) that, given encryptions c_1 and c_2 of m_1 and m_2, outputs a ciphertext encrypting the product $m_1 \cdot m_2$.

B. Encryption parameters

The specific instantiation that is implemented in SEAL is the more practical variant of the scheme YASHE, proposed in the paper [1]. The encryption parameters of the scheme are the degree n, the moduli q and t, the decomposition word size w, and distributions χ_{key},χ_{err}. Thus, parms := (n,q,t,w,χ_{key},χ_{err}). Next we discuss these parameters in more detail.

- The parameter n is the maximum number of terms in the polynomials used to represent both plaintext and ciphertext elements. In SEAL, n is always a power of 2. The polynomial $X^n + 1$ is called the *polynomial modulus*, and denoted poly_modulus in SEAL.
- The parameter q is the *coefficient modulus*. It is an integer modulus used to reduce the coefficients of ciphertext polynomials. In SEAL, q is called coeff_modulus.
- The parameter t is the *plaintext modulus*. It is an integer modulus used to reduce the coefficients of plaintext polynomials. In SEAL, t is called plain_modulus.
- The integer w is the base to which we decompose integer coefficients into smaller parts. It determines the number $\ell_{w,q} := \lfloor \log_w(q) \rfloor + 1$ of parts when decomposing an integer modulo q to the base w. In practice, we take w be a power of two, and call $\log_2 w$ the *decomposition bit count*. In SEAL, $\log_2 w$ is called decomposition_bit_count.
- The distribution χ_{key} is a probability distribution on polynomials of degree at most $n - 1$ with integer coefficients, which is used to sample polynomials with small coefficients that are used in the key generation procedure. In SEAL, coefficients are sampled uniformly from $\{-1,0,1\}$.
- Similarly, the distribution χ_{err} on polynomials of degree at most $n - 1$ is used to sample noise polynomials, which are needed during both key generation and encryption. In SEAL, the distribution χ_{err} is a truncated discrete Gaussian centered at zero, with standard deviation σ. In SEAL, σ is called noise_standard_deviation.

The remainder of this subsection goes into further detail, introduces the necessary mathematical structures, and explains how the different parameters are related to each other. The scheme operates in the ring $R := \mathbb{Z}[X]/(X^n + 1)$, the elements of which are polynomials with integer coefficients of degree less than n, where n is a power of 2. Any element $a \in R$ can be written as $a = \sum_{i=0}^{n-1} a_i X^i$, with $a_i \in \mathbb{Z}$. All plaintexts, ciphertexts, encryption and decryption keys, etc. are elements of the ring R, and have this form. Addition in R is done coefficient-wise, and multiplication is simply polynomial multiplication modulo $X^n + 1$, i.e. standard polynomial multiplication followed by reduction modulo X^n+1. The reduction modulo $X^n + 1$ is carried out by replacing all occurrences of X^n by -1.

The scheme uses two integer moduli q and t, for which q is much larger than t. The coefficients of ciphertext and key elements are taken modulo q, whereas the

coefficients of the plaintext elements are taken modulo t. In what follows, we use the notation $[a]_q$ (resp. $[a]_t$) to denote the operation of reducing the coefficients of $a \in R$ modulo q (resp. t) into the set $\{\lceil -q/2 \rceil, \ldots, \lfloor (q-1)/2 \rfloor\}$ (resp. $\{\lceil -t/2 \rceil, \ldots, \lfloor (t-1)/2 \rfloor\}$).

The homomorphic multiplication operation, defined below, contains a step which decomposes a given polynomial into a vector of polynomials with smaller coefficients. This step is needed to manage the noise growth during the homomorphic multiplication by computing a product of two intermediate polynomials via a scalar product. The size of the smaller coefficients is determined by the parameter w, which thus controls a tradeoff between multiplication efficiency and evaluation key size on one hand, and noise growth on the other. We now present the details for this decomposition.

A polynomial $a \in R$ with coefficients in $\{\lceil -q/2 \rceil, \ldots, \lfloor (q-1)/2 \rfloor\}$ can be decomposed using a base $w \in \mathbb{Z}$ as $a = \sum_{i=0}^{\ell_{w,q}-1} a_i w^i$, where the $a_i \in R$ have coefficients in $\{\lceil -w/2 \rceil, \ldots, \lfloor (w-1)/2 \rfloor\}$. This is done by decomposing each coefficient to the base w. The homomorphic encryption scheme makes use of two functions. The first one is $\mathrm{Dec}_{w,q}(a) := ([a_i]_w)_{i=0}^{\ell_{w,q}-1}$, which takes a polynomial with coefficients modulo q, and returns the vector of polynomial parts obtained by the w-adic decomposition described above. The second one is $\mathrm{Pow}_{w,q}(a) := ([aw^i]_q)_{i=0}^{\ell_{w,q}-1}$, which takes a polynomial and returns a vector of polynomials that are the products of the polynomial with powers of the base w. Both functions take a polynomial and map it to a vector of polynomials in $R^{\ell_{w,q}}$, such that the following property holds $\langle \mathrm{Dec}_{w,q}(a), \mathrm{Pow}_{w,q}(b) \rangle = a \cdot b \pmod q$,

where $\langle \cdot, \cdot \rangle$ denotes the dot product of vectors (of polynomials), defined in the usual way.

Finally, the scheme uses two probability distributions on R, χ_{key} and χ_{err}, which both generate polynomials in R with small coefficients. In our implementation, we chose the distribution χ_{key} as the uniform distribution on polynomials with coefficients in $\{-1,0,1\}$. Sampling an element according to this distribution means sampling all its coefficients uniformly from $\{-1,0,1\}$. For the distribution χ_{err}, we use a discrete Gaussian distribution with mean 0 and appropriately chosen standard deviation σ. Gaussian samplers typically sample from a truncated discrete Gaussian distribution, and we denote the bound, i.e. the maximal deviation from the mean (zero), by B_{err}. A typical large enough choice for B_{err} would be around 5σ.

C. Plaintext space and homomorphic operations

All plaintext elements, i.e. the messages that can be encrypted with the homomorphic encryption scheme, are polynomials in the ring R, with coefficients reduced modulo the integer t. All ciphertext elements, i.e. encryptions of plaintext elements, are polynomials in the ring R, with coefficients reduced modulo the integer q. Formally, this means that the plaintext space is the ring $R_t := R/tR \tilde{=} \mathbb{Z}_t[X]/(X^n+1)$, and the ciphertext space is contained in the ring $R_q := R/qR \tilde{=} \mathbb{Z}_q[X]/(X^n+1)$. However, not every element of R_q is a valid ciphertext.

Any ciphertext produced by the encryption function of our scheme, as described below, encrypts one plaintext message polynomial $m \in R_t$. Whenever homomorphic addition (resp. multiplication) is performed on ciphertexts that encrypt two plaintext elements, say $m_1, m_2 \in R_t$, the resulting ciphertext will encrypt the sum m_1+m_2 (resp. the product $m_1 \cdot m_2$). The operations between the plaintext elements are performed in the ring R_t.

For homomorphic addition this means that the resulting ciphertext will encrypt the coefficient-wise sum m_1+m_2, where the coefficients are automatically reduced modulo the plaintext modulus t. For homomorphic multiplication the resulting ciphertext will encrypt the product $m_1 \cdot m_2 \in R_t$, which means that the polynomial will automatically be reduced modulo X^n+1, i.e. all powers X^n will be automatically replaced by -1, until no monomials of degree n or higher remain, and just as in homomorphic addition, the coefficients of the polynomial $m_1 \cdot m_2$ will be automatically reduced modulo t.

These properties need to be taken into account when encrypting data such as integers or real numbers that first need to be encoded as plaintext polynomials. One needs to be aware of the fact that the various reductions that occur on plaintext polynomials during homomorphic operations do not necessarily correspond to meaningful operations on the integral or real data.

D. Detailed algorithm description

The following gives a detailed description of the key generation, encryption, decryption, and homomorphic evaluation algorithms.

- KeyGen(parms): On input the encryption parameters parms := $(n,q,t,\chi_{\text{key}},\chi_{\text{err}})$, the key generation algorithm samples polynomials $f^0,g \leftarrow \chi_{\text{key}}$ from the key distribution, and sets $f := [1 + tf^0]_q$. If f is not invertible modulo q, it chooses a new f^0. Otherwise, it computes the inverse f^{-1} of f in R_q. Next, the algorithm samples vectors $\mathbf{e},\mathbf{s} \in R^{w,q}$, for which each component is sampled according to the error distribution χ_{err}, and computes the vector of polynomials $\gamma := [\text{Pow}_{w,q}(f) + \mathbf{e} + h\mathbf{s}]_q$. It computes $h := [tgf^{-1}]_q \in R$, and outputs the key pair
$$(\text{pk},\text{sk}) := (h,f) \in R \times R,$$
and the evaluation key evk := γ.
- Enc(h,m): To encrypt a plaintext element $m \in R_t$, the encryption algorithm samples small error polynomials $s,e \leftarrow \chi_{\text{err}}$, and outputs the ciphertext $c := [\text{b}^q/_t\text{c}m + e + hs]_q \in R$.
- Dec(f,c): Given the private decryption key f, and a ciphertext c = Enc(h,m), the decryption algorithm recovers m using $m = [\text{b}t/_q \cdot [fc]_q\text{e}]_t \in R$.
- Add(c_1,c_2): Given two ciphertexts c_1 and c_2, the algorithm Add outputs the ciphertext $c_{\text{add}} := [c_1 + c_2]_q$.
- Mult(c_1,c_2,evk): Given two ciphertexts c_1 and c_2, the algorithm Mult first computes $\tilde{c}_{\text{mult}} := [\text{b}t/_q(c_1 \cdot c_2)\text{e}]_q$. It then performs a so-called *relinearization* (or *key switch*) operation, by returning $c_{\text{mult}} := [\text{hDec}_{w,q}(\tilde{c}_{\text{mult}}),\text{evki}]_q$.

E. Practical considerations

As we already explained in the beginning of Section II, every ciphertext, even a freshly encrypted one, has a certain amount of inherent noise, or error, in it. The decryption operation can be understood as an algorithm for removing this noise using some auxiliary information, namely the secret key. One of the main difficulties in homomorphic cryptography is that in every homomorphic operation this inherent noise increases, until it reaches its maximum, at which point the message becomes so distorted that even the decryption algorithm can not recover it. To counter this problem, one needs to increase the parameter q, but for security reasons this means that also n should be increased. Unfortunately, increasing n and q can significantly degrade performance.

There are a number of ways to lower the noise growth during homomorphic operations, at least in certain situations, and thus to improve performance by allowing smaller parameters to be used. For example, the function to be computed might involve publicly known values that do not need to be encrypted before adding them to, or multiplying them with an encrypted value. One simply needs to mimic the standard operations described above, and include the public values as ciphertexts obtained from an encryption procedure in which all noise terms are set to zero. This approach yields significantly smaller noise growth, allowing for the same number of homomorphic operations to be performed with smaller parameters, and thus will indirectly lead to improved performance.

Furthermore, when such public values are small, a multiplication with them can be made much more efficient by using their representation according to the currently used encoding technique. The multiplication can then be performed by a sequence of shifts (multiplications by powers of X) and homomorphic additions, avoiding the multiplication routine altogether. Since typically homomorphic additions are significantly less costly than a homomorphic multiplication, this approach can increase the efficiency of a computation considerably. Due to their importance, SEAL contains functionality for performing addition and multiplication by a (public) plaintext polynomial.

Another promising avenue is to omit the relinearization step (recall the description of Mult in II-D). The homomorphic multiplication algorithm then only computes the polynomial $\tilde{c}_{\text{mult}}$. Thus, we can replace Mult by

- Mult$_{\text{norelin}}(c_1,c_2)$: Given two ciphertexts c_1 and c_2, the algorithm returns $\tilde{c}_{\text{mult}} := [\text{b}t/_q(c_1 \cdot c_2)\text{e}]_q \in R$.

The result of this operation does not give the correct value when decrypted with the secret key sk, but instead needs to be decrypted with the square of the secret key, $[\text{sk}^2]_q \in R$. Further multiplications of this kind increase the required power of the secret key. This means that the decryption algorithm needs to be called with the corresponding power s of the secret key, and now looks as follows:

- $\mathrm{Dec}_{\mathrm{norelin}}(f,s,c)$: Given the private decryption key f, an exponent s, and a ciphertext c, the decryption algorithm recovers m using $m = [\lfloor t/q \cdot [f^s c]_q \rceil]_t \in R$.

This approach has the advantage that it omits the by far most costly part of the homomorphic evaluation algorithms, and works without the evaluation key. Its usefulness, however, depends on the specific choice of the encryption parameters. For example, the larger the plaintext modulus t is, the fewer levels of multiplications can be computed like this, given all other parameters stay fixed. One can experiment with different trade-offs, for example by manually reintroducing relinearization steps at certain points in the computation.

F. Implementation

We now demonstrate how the above concepts are implemented in SEAL. Here we present mostly code snippets, and for complete examples we refer the reader to Section VI.

SEAL is written in C++, but comes with a C# wrapper library SEALNET. All of our code examples use the C++ library. The necessary C++ header files are included with **#include** "seal.h". SEAL contains a data type BigUInt for large unsigned integers, and a data type BigPoly for large polynomials with BigUInt coefficients. All polynomials used in the encryption scheme are stored using instances of BigPoly, including plaintext and ciphertext polynomials, the secret key, and the public key. For example, to create a (plaintext) polynomial $p(X) = 3X^4 + X + 2$, we can write

```
BigPoly p("3x^4 + 1x^1 + 2");
```

To access the i-th coefficient as a BigUInt, we can use p[i]. To return the polynomial as a human-readable string, we can use the member function to_string. We have seen above that plaintext polynomials can have either positive or negative numbers as coefficients, but in SEAL the coefficients of BigPoly are always instances of BigUInt, i.e. unsigned. For this reason we store coefficients in the range $\{\lceil -t/2 \rceil, \ldots, -1\}$ instead as $\{\lfloor (t-1)/2 \rfloor + 1, \ldots, t-1\}$. For example, if $t = 2^{10}$ ($0x400$ in hexadecimal), the polynomial $p(X) = X^2 - 3X - 1$ could be created using

```
BigPoly p("1x^2 + 3FDx^1 + 3FF");
```

To set up the cryptosystem, the first thing we must do is choose the encryption parameters as described in II-B. These are encapsulated in an instance of the class EncryptionParameters. First, we set the three moduli that the encryption scheme uses: q (coefficient modulus), t (plain modulus), and $X^n + 1$ (polynomial modulus). These three are the most important parameters, and choosing them correctly is crucial for achieving optimal performance. Next we set w, or rather the decomposition bit count $\log_2 w$, the standard deviation σ of the distribution χ_{err}, and an upper bound for the output of the χ_{err} sampler. For the purpose of the examples here, we use the following parameters:

Listing II.1. encryption_parameters

```
EncryptionParameters parms; parms.poly_modulus() = "1x^2048 +
1"; parms.coeff_modulus() = ChooserEvaluator::
default_parameter_options().at(2048);
parms.plain_modulus() = 1 << 10; parms.decomposition_bit_count() =
32; parms.noise_standard_deviation() = ChooserEvaluator::
default_noise_standard_deviation();
parms.noise_max_deviation() = ChooserEvaluator::
   default_noise_max_deviation();
```

In the above, coeff_modulus and plain_modulus are instances of BigUInt, poly_modulus is an instance of BigPoly, decomposition_bit_count is an **int**, and the last two are **double**. In general, choosing appropriate and optimal encryption parameters is a surprisingly difficult task. For this reason SEAL provides an easy-to-use automatic parameter selection module, which we discuss in more detail in Section IV. A part of this can be seen in the above where coeff_modulus, noise_standard_devation and noise_max_deviation are set to values hard-coded into the library that we consider secure.

Next we need to generate the encryption keys. Specifically, there are three types of keys we need to set: the public key, the secret key, and the evaluation key (recall II-D). Of these, the public key and the secret key are instances of BigPoly, and the evaluation key is encapsulated in an EvaluationKeys object. The keys are generated based on an instance of EncryptionParameters using KeyGenerator as follows:

Listing II.2. key_generator

```
KeyGenerator generator(parms); generator.generate();
BigPoly public_key = generator.public_key();
BigPoly secret_key = generator.secret_key(); EvaluationKeys
evaluation_keys = generator. evaluation_keys();
```

The next set of tools we need are for encrypting, decrypting, and performing homomorphic operations:

Listing II.3. encryption_tools

```
Encryptor encryptor(parms, public_key);
Decryptor decryptor(parms, secret_key); Evaluator evaluator(parms,
evaluation_keys);
```

The following code constructs two plaintext polynomials, $X^2 - 2X$, and $3X^3 + 1$, and encrypts them:

```
BigPoly plain1("1x^2 + 3FEx^1");
BigPoly plain2("3x^3 + 1");
BigPoly enc1 = encryptor.encrypt(plain1);
BigPoly enc2 = encryptor.encrypt(plain2);
```

We can use the Evaluator to operate on ciphertext polynomials:

```
BigPoly enc_sum = evaluator.add(enc1, enc2); BigPoly enc_prod =
evaluator.multiply(enc1, enc2);
```

In addition to add and multiply, Evaluator supports a number of other operations. For example, it is very efficient to add and multiply ciphertexts by plaintext polynomials (recall II-E):

```
BigPoly p("1x+2"); // Public polynomial
BigPoly enc1_sum = evaluator.add_plain(enc1, p);
BigPoly enc2_prod = evaluator.multiply_plain(enc2,
p);
```

The above code produces encryptions of $(x^2 - 2x) + (x + 2)$ and $(3x^3 + 1) \cdot (x + 2)$. Finally, we can decrypt using our instance of Decryptor:

```
BigPoly sum = decryptor.decrypt(enc_sum);
BigPoly prod = decryptor.decrypt(enc_prod);
BigPoly plain_sum = decryptor.decrypt(enc1_sum); BigPoly plain_prod
= deryptor.decrypt(enc2_prod);
```

SEAL also supports multiplication without relinearization, and a stand-alone relinearization operation (recall II-E). These are provided by the member functions multiply_norelin and relinearize of Evaluator.

III. Encoding Data

As was described in II-B and II-C, plaintexts and ciphertexts are certain polynomials rather than integers or real numbers. More precisely, plaintext elements are polynomials in R, with coefficients reduced modulo t, and ciphertexts are polynomials in R, with coefficients reduced modulo q. However, most algorithms in genomics and bioinformatics operate on integers, real numbers, and binary values. Therefore, there is a mismatch between the plaintexts used by the encrypt function in SEAL, and the data types used by practitioners. This mismatch is resolved using *encodings*, which convert common data types into plaintext polynomials. Encodings must always come with a matching *decoding*, which performs the inverse operation. For the homomorphic property of the encryption scheme to make sense, the encoding and decoding functions must also be homomorphic in such a way that addition (resp. multiplication) of encoded plaintext polynomials yields an encoding of the sum (resp. product) of the encoded integers or real numbers.

The main challenges in designing an encoding are (1) making sure that the encoding and decoding functions have the appropriate homomorphic poperties (see above), and (2) making sure that the representation is compact and allows for fast and memory efficient computation. As a simple example, consider encoding an integer as the scalar coefficient of a plaintext polynomial. Decoding in this case is trivial: Simply read the constant coefficient. However, if at any point during the computation the values of the constant coefficient increase beyond t, it will automatically be reduced modulo t, and the result might be unexpected. The solution is to choose t large enough, but this might in turn cause the inherent noise to grow very rapidly in homomorphic multiplications. When encoding integers or real numbers as higher degree polynomials, it is typically necessary to keep track of the degrees of the plaintext polynomials appearing during the computation, since if they exceed X^{n-1} reduction modulo the polynomial modulus $X^n + 1$ might occur, again leading to unexpected results.

In the following section we present several powerful encoding techniques. The choice of the right encoding depends on the problem. Moreover, there are many other encoding techniques that might be more appropriate for certain applications.

A. Encoding integers

The simplest way to encode an integer is what we already mentioned above: by representing it as the constant coefficient of a plaintext polynomial. Therefore, an integer y is encoded as the constant polynomial $p(X) := y$. This *scalar encoding* works as long as the numbers used during the computation remain in $\{\lceil -t/2 \rceil, \ldots, \lfloor (t-1)/2 \rfloor\}$. Otherwise reduction modulo t might occur, yielding unexpected results.

The scalar encoding has two main limitations: (1) large integers cannot be encoded without choosing t to be enormous, and (2) it is inefficient in its use of available space in the plaintext polynomial $p(X)$, which has a total of n coefficients waiting to be used. For one way to resolve both (1) and (2), consider the following.

Let $\sum_i b_i 2^i$ be the binary representation of an integer y. We can encode y as the polynomial $p(X) := \sum_i b_i X^i$. In this case the number y can be recovered using $y = p(2)$, so this encoding also admits an efficient and straightforward decoding. The advantage of this *binary encoding* over the scalar encoding is that the coefficients at the beginning of each computation have only small values: either zero or one. While adding and multiplying may increase the sizes of the coefficient, they will still grow much slower than in the scalar encoding, and therefore may not so easily reduced modulo t. However, modular reduction may happen, as may reduction modulo around X^n+1, which was not possible with the scalar encoding. A negative number we would encode by simply negating every coefficient of the binary expansion of its absolute value.

The binary encoding uses a base-2 representation to encode integers, but we can just as well use any higher base b, although this comes with the cost of having larger coefficients appear in the freshly encoded plaintext polynomial. Consider an odd base $b \geq 3$. In this case we can use a *balanced* set of representatives of integers modulo b as coefficients of the base-b representation. In other words, each integer has a unique base-b representation, where powers of b appear with coefficients from the symmetric range $\{\lfloor -b/2 \rfloor, \ldots, \lfloor b/2 \rfloor\}$. Encoding using the binary encoding wastes space since each coefficient is one of $\{-1,0,1\}$, but all non-zero ones will necessarily have the same sign. In balanced base-3 representation each coefficient again belongs to the set $\{-1,0,1\}$, but now they can have different signs. For example, encoding the number 25 using balanced base-3 encoding would yield the polynomial $p(X) := X^3 - X + 1$. Decoding amounts to evaluating the polynomial at $X = 3$. Using a higher base b produces shorter polynomials with larger coefficients. This might be advantageous if the numbers to be encoded are very large.

Another way to handle large numbers is by encoding them multiple times using several co-prime plaintext moduli $t_1,\ldots,t_k$. Decoding can be done using the Chinese Remainder Theorem (CRT)[2] to combine the individual decodings back into an integer modulo $\prod t_i$. Therefore, we can break a large integer y into k much smaller integers, obtained by reducing y modulo the t_i, and for instance use the scalar encoding to these separately, encrypt them, and operate on them. Note that each operation must now be performed on each of the k ciphertexts. Once done with the computation, use CRT to assemble the decrypted remainders into a single number modulo t. This method is obviously incredibly wasteful in terms of space, but allows for very small t (and hence also q and n) to be used with the scalar encoding. Note that the range of integers that can be encoded increases exponentially with k.

SEAL provides classes BinaryEncoder and BalancedEncoder that contain all of the essential functionality for using the binary and balanced odd base encodings. The following C++ code demonstrates encoding the integer 1234 using binary, balanced base-3 and balanced base-5 encodings.

```
encryption_parameters (Listing II.1) key_generator (Listing
II.2) encryption_tools (Listing II.3)

BinaryEncoder encoder2(plain_modulus);

/* In BalancedEncoder the base defaults to 3 */
BalancedEncoder encoder3(plain_modulus);
BalancedEncoder encoder5(plain_modulus, 5);

BigPoly e2 = encoder2.encode(1234);
BigPoly e3 = encoder3.encode(1234);
BigPoly e5 = encoder5.encode(1234);
```

The encoded BigPoly objects can be printed, as usual, using the member function to_string.

B. Encoding real numbers

The most straightforward way to encode reals is by scaling them to integers, which of course can only be done when a fixed precision is needed. For example, suppose we are given real numbers for which 3 digits after the decimal point are significant. Multiplying these numbers by 1000 and ignoring the fractional part will results in integer values that capture the significant information. Note however, that some book-keeping is required since all the results will be scaled up. Moreover, when multiplying numbers that were scaled up, the result will have a different scaling factor than the inputs. As a result, it is important to keep track of the scale at different parts of the calculation, which can get rather tedious with complicated computations.

[2] According to CRT, given co-prime integers $t_1,\ldots,t_k$, and integers $r_i \pmod{t_i}$, there is exactly one integer $y \pmod{\prod t_i}$ such that $y \equiv r_i \pmod{t_i}$ for every i.

When the number of significant digits is large, the above method will result in very large integers to be encoded. This can be avoided by a different encoding scheme, where we encode the digits after the decimal point as the highest degree coefficients of the polynomial. More precisely, a real number $y = y^+.y^-$, where y^+ denotes the binary digits $b_{l^+}b_{l^+-1}\ldots b_1b_0$, and y^- denotes the binary digits $b_{-1}b_{-2}\ldots b_{-l^-}$, is encoded as the plaintext polynomial

$$\sum_{i\le l^+} X_i b - \sum_{0<i\le l^-} X_{n-i} b_{-i}.$$

For example, 2 will be encoded as X, while $1/2$ is encoded as $-X^{n-1}$. When these two representations are multiplied, we obtain

$$X \cdot (-X^{n-1}) = -X^n = 1 \pmod{X^n + 1},$$

as should be expected.

As another simple example, consider $3.25 = 11.01_2$ encoded as $-X^{n-2} + X + 1$. Multiplying this by $1.5 = 1.1_2$ encoded as $-X^{n-1} + 1$ gives $(-X^{n-2} + X + 1)(-X^{n-1} + 1)$

$$= X^{2n-3} - X^{n-2} - X^n - X^{n-1} + X + 1$$

$$= -X^{n-1} - X^{n-2} - X^{n-3} + X^2 \pmod{X^n + 1}.$$

Decoding yields $2^2 + 2^{-1} + 2^{-2} + 2^{-3} = 4.875$, which is what we would expect.

When setting up such a fractional encoder, we need to tell how many plaintext polynomial coefficients are reserved for the fractional part, and how many are reserved for the integral part, because the encoding algorithm needs to know where to truncate a possibly infinite expansion of the fractional part, and the decoding algorithm needs to know which coefficients belong to which part, as they must be treated differently. For example, if we have $n = 4096$, we could reserve 1024 coefficients for the integral part, and 128 for the fractional part. Freshly encoded numbers can then use all of the 128 highest coefficients for their fractional parts, and up to 1024 lowest coefficients for their integral parts. When two such polynomials are multiplied, they can have up to 256 of their top coefficients be non-zero. Clearly when such polynomials are further multiplied, the fractional part quickly grows down towards the coefficients reserved for the integral part. In this case the fractional part can take up to $4096 - 1024 = 3072$ coefficients, but after that it gets mixed with the integral part and can yield unexpected results when decoded. In the decoding process we would only count the 128 highest coefficients towards the fractional part, the lowest 1024 towards the integral part, and ignore the rest. Our library provides basic fractional encoding functionality in the BinaryFractionalEncoder and BalancedFractionalEncoder classes.

We present an example of computing $3.14 \cdot 15.93$ in encrypted form. In this example we use fractional balanced base-3 encoding, reserve 128 coefficients for the fractional part, and 256 coefficients for the integral part.

```
encryption_parameters (Listing II.1) key_generator (Listing
II.2) encryption_tools (Listing II.3)

/* The base defaults to 3 */
BalancedFractionalEncoder encoder(parms. plain_modulus(),
   parms.poly_modulus(), 256,
128);

BigPoly plain1 = encoder.encode(3.14);
BigPoly plain2 = encoder.encode(15.93);

BigPoly enc1 = encryptor.encrypt(plain1);
BigPoly enc2 = encryptor.encrypt(plain2);

BigPoly enc_prod = evaluator.multiply(enc1, enc2);

BigPoly prod = decryptor.decrypt(enc_prod); double result =
encoder.decode(prod);
```

The correct answer, stored in result, is 50.0202.

C. Plaintext packing

SEAL requires working with high degree polynomials to ensure both security and correctness of the computation. However, the data to be encoded is in many cases rather small, resulting in an enormous message expansion rate, and relatively long encoding/encryption/decryption times. One way to avoid these problems is to pack several pieces of data in a single message, and use the Single Instruction Multiple Data (SIMD) paradigm to operate on these messages [5], [29].

One way to encode more data in a single message is by using the Chinese Reminder Theorem (CRT) for polynomial rings. For example, suppose that $n = 2$ and $t = 5$. Since $X^2 + 1 = (X + 2)(X + 3) \pmod 5$, CRT yields an explicit isomorphism

$$\frac{\mathbb{Z}_5[X]}{(X^2+1)} \cong \frac{\mathbb{Z}_5[X]}{(X+2)} \times \frac{\mathbb{Z}_5[X]}{(X+3)}.$$

This isomorphism allows taking two values, one in $Z_5[X]/(X + 2)$, and the other in $Z_5[X]/(X + 3)$, and encoding them as a single element in $Z_5[X]/(X^2 + 1)$. More generally, assume that

$$X^n + 1 = \prod_{i=1}^{k} Q_i(X) \pmod{t}$$

for some polynomials $Q_1(X),\ldots,Q_k(X)$ that are coprime[3]. Then

$$\frac{\mathbb{Z}_t[X]}{(X^n+1)} \cong \prod_{i=1}^{k} \frac{\mathbb{Z}_t[X]}{(Q_i(x))} \pmod{t}.$$

This allows to encode k integers in a single plaintext as, for instance, the constant coefficients of each of the k factors, and to operate on each of them simultaneously.

[3] This means that if $R(X)$ is a polynomial that divides both $Q_i(X)$ and $Q_j(X)$ such that $i \neq j$, then $R(X)$ is a constant.

In many cases it is possible to find n such polynomials $Q_1(X),\ldots,Q_n(X)$ which are co-prime, and $X^n + 1 = \prod Q_i(X) \pmod{t}$, in which case, each $Q_i(X)$ must be a linear polynomial, $Z_t[X]/(Q_i(X)) \tilde{=} Z_t$, and $Z_t[X]/(X^n + 1) \tilde{=} Z_t^n$. This is the optimal case, and allows encoding of n integers into one plaintext polynomial. Of course this only makes sense if the scalar encoding is otherwise appropriate for the problem at hand.

Plaintext packing is implemented in SEAL in the class PolyCRTBuilder.

D. Encoding binary data

In some situations encrypting integers is more convenient to do *bit-by-bit*. This is particularly useful for efficient comparison or equality testing, but is not very efficient or practical when homomorphic multiplication of integers is required. For example, one situation where bit-wise encryption is particularly useful is in computing the *edit distance* between two short encrypted DNA sequences [11]. One option is to use the scalar encoding together with a plaintext modulus $t = 2$, so that the plaintexts are elements of $Z_2[X]/(X^n + 1)$. This is of course incredibly inefficient in many ways, but it does allow the user to perform both XOR and AND operations on individual bits using homomorphic addition and multiplication, respectively, providing an enormous amount of functionality.

A na¨ıve way to improve the performance is by encoding up to n bits as the n coefficients of a plaintext polynomial. This does allow for some parallelism, namely one can evaluate XOR gates on n encrypted pairs of bits with just one homomorphic addition, or to evaluate either XOR or AND with a plaintext bit on n encrypted bits simultaneously. What is not possible however, is evaluating AND gates on n encrypted pairs of bits simultaneously.

A much better way to introduce parallelism to bitwise encryption is to use the CRT technique of III-C. In this case up to n bits can be encoded in the constant coefficients of the different *slots*, where both addition and multiplication are performed separately for each slot, resulting in massive improvements in the amortized complexity.

IV. Parameter selection

Selecting secure parameters for homomorphic encryption schemes is a surprisingly complicated task. Security of the encryption scheme used by SEAL depends on the assumed hardness of a lattice problem known as *Ring-Learning With Errors (RLWE)* [1], [22]. In some parameter settings the hardness of RLWE can further be proven to that of certain extremely well studied worst-case lattice problems [28], [23], [26], [24], [7], but unfortunately such parameters are not relevant for practical use. Instead, in practice the security claims must be directly based on an analysis of state-of-the-art attacks against RLWE, which has been done in [20], [32], [21].

In addition to guaranteeing security, the encryption parameters must also be *large* enough, or else the inherent noise (recall Section II) will grow too big, and make the ciphertexts impossible to decrypt. We denote $\Delta := \lfloor q/t \rfloor$, and by $r_t(q)$ the (positive) remainder when dividing q by t. The inherent noise in a ciphertext $c \in R$ is a polynomial $v \in R$, such that

$$fc \equiv \Delta[m]_t + v \pmod{q}.$$

It is shown in [1] that a ciphertext c is possible to decrypt only as long as it has an inherent noise that satisfies

$$||v||_\infty < \frac{\Delta - r_t(q)}{2}.$$

Here $||v||_\infty$ denotes the largest absolute value of the coefficients of v. Even freshly encrypted ciphertexts have a certain amount of noise in them (see IV-A below), and performing arithmetic operations on ciphertexts always

increases the noise level, until it reaches its maximum value and corrupts the underlying plaintext. Due to the significance of $||v||_\infty$, we often call it the inherent noise, instead of v. In fact, our noise growth simulator only estimates the growth of $||v||_\infty$, rather than of v.

In addition, the encoding scheme typically places strong restrictions on the size of the plaintext modulus t, and in some cases on the degree n of the polynomial modulus, as was explained in Section III. Hence, to set up the cryptosystem with appropriate parameters, the user must perform (roughly) the following steps:

(1) Let σ be a constant or possibly a function of n.
(2) Determine lower bounds for t and n (depending on encoding).
(3) Determine a lower bound for q such that decryption can be *expected* to succeed.
(4) Choose n, q, and possibly σ, based on the bounds determined above and state-of-the-art security estimates.
(5) Choose w to be as large as possible (at most q) so that decryption still succeeds.
(6) If w chosen above is too small, or no such w could be chosen, switch to larger n and q.

Our automatic parameter selection tool essentially performs the above steps. By default it uses a constant $\sigma := 3.19$, a constant bound $B_{\text{err}} = 15.95$ for the Gaussian error sampler, and a hard-coded list of pairs (n,q) that we consider to be secure based on the analysis of [20]. Table IV shows the size of q for the values of n that are used by the automatic parameter selector. A noise growth simulator then estimates the growth of inherent noise in the homomorphic operations without requiring any actual encrypted data as input. These default values are all available also outside the automatic parameter selector through the static functions

default_noise_standard_deviation()
default_noise_max_deviation()
default_parameter_options()

of ChooserEvaluator. Alternatively, the user can supply their own σ, B_{err}, and list of (n,q)-pairs for the parameter selector to use.

TABLE I
(n,q)-PAIRS

n	1024	2048	4096	8192	16384
q	48 bits	91 bits	127 bits	383 bits	768 bits

A. Noise growth simulator

Upper bounds for inherent noise growth in the homomorphic encryption scheme are well understood [1], and have already been used for parameter selection in e.g. [2]. The problem with these bounds is that they are typically extremely conservative, and as such yield highly inefficient parameters. We instead focus on the most significant terms contributing to the noise, and use average-case estimates for their sizes. As a result, we obtain simple, but fairly accurate estimates for inherent noise growth in all homomorphic operations. Moreover, these estimates only require the encryption parameters, and the inherent noise estimates for the input ciphertexts to work. More precisely, given input ciphertexts with estimated inherent noises v_1,v_2, the estimated inherent noise of the output is computed as follows:

fresh: $\sqrt{\frac{2n}{3}}\, B_{\text{err}}\, 2t$

add: $||v_1||_\infty + ||v_2||_\infty$ multiply:

$$\sqrt{\frac{2n}{3}}\left[\frac{nt^2}{2}\left(||v_1||_\infty + ||v_2||_\infty\right) + \sqrt{n}\,\ell_{w,q}\,B_{\text{err}}\,w\,t\right]$$

add_plain: $||v||_\infty$

multiply_plain by $p(X)$: $\deg(p)+1)||v||_\infty\,||p||_\infty$ negate: $||v||_\infty$

The above estimates are only valid when the parameters are in realistic ranges, and only until the inherent noise reaches its upper bound of $(\Delta - r_t(q))/2$. It is crucial to understand that both homomorphic addition and multiplication by a plaintext polynomial typically increase the noise significantly less than true homomorphic multiplication of two ciphertexts, which can be easily seen from the estimates.

Recall from II-E that it is possible to also do multiplication without performing the relinearization procedure, but that the result must then be decrypted with a different secret key. In the above noise growth estimate for multiply, the first term comes from the multiply_norelin part, and the second from the operation relinearize.

To understand what is involved in these estimates, consider for example estimating the noise in a freshly encrypted ciphertext. Recall that the encryption of a plaintext polynomial $m \in R$ is $c := [\lfloor q/t \rfloor c[m]_t + e + hs]_q \in R$,

where $h := [tgf^{-1}]_q$ is the public key, and $s,e \leftarrow \chi_{\text{err}}$. To find the inherent noise in c, we compute

$$[fc]_q = [(1+tf')(\lfloor q/t \rfloor [m]_t + e + hs)]_q$$
$$= [\lfloor q/t \rfloor [m]_t + fe + tgs - r_t(q)\, f'\, [m]_t]_q,$$

where coefficients of the secret polynomial f^0 are chosen uniformly at random from $\{-1,0,1\}$. We have also used $q = \lfloor q/t \rfloor t + r_t(q)$. The inherent noise polynomial in a fresh ciphertext is therefore

$$v_{\text{fresh}} := [fe + tgs - r_t(q)\, f'\, [m]_t]_q.$$

It is the $||\cdot||_\infty$-norm of v_{fresh} that matters, and that we need to estimate. By the triangle inequality

$$||v_{\text{fresh}}||_\infty \le ||fe||_\infty + ||tgs||_\infty + r_t(q)||f^0\,[m]_t||_\infty.$$

In typical cases the last term is very small compared to the first two, so we omit it. This is especially true when $r_t(q)$ is small, when $[m]_t$ has small coefficients, and/or when $[m]_t$ is sparse and short enough. In the first term, the polynomial e is dense and has coefficients distributed according to a discrete Gaussian distribution with mean 0 and standard deviation σ. Since approximately two thirds of the coefficients of f are $\pm t$, and one third are 0, each coefficient of fe is roughly a sum of $2n/3$ discrete Gaussian distributions, multiplied by t, and hence distributed according to a discrete Gaussian distribution with standard deviation $t\sqrt{2n/3}\sigma$. This means that we can expect $||fe||_\infty \approx t\sqrt{2n/3}B_{\text{err}}$. The polynomial tgs has similar structure, so also $||tgs||_\infty \approx t\sqrt{2n/3}B_{\text{err}}$, resulting in the estimate above for fresh. The other ones involve similar approximations.

B. Automatic parameter selection

As we explained in the beginning of this section, automatic parameter selection involves much more than simply estimating inherent noise growth. In particular, the plaintext modulus t must be large enough for decoding to work. The user must provide an estimated size of the input plaintext data, in particular bounds on the lengths of the plaintext polynomials and on the absolute values of their coefficients, and of course the homomorphic operations that are to be performed. From all this information we can compute a compute a lower bound for t. The homomorphic operations must also be stored in order to later run the noise simulator.

In practice we perform this by introducing a device called ChooserPoly. These objects carry three essential pieces of data:

(CP1) Upper bound on the number of non-zero coefficients in a (plaintext) polynomial

(CP2) Upper bound on the $||\cdot||_\infty$-norm of a (plaintext) polynomial

(CP3) A directed acyclic graph representing the entire *operation history* of the ChooserPoly

One should think of a ChooserPoly as modeling a ciphertext, while only carrying information about the size of the underlying plaintext polynomial, and information about how the ciphertext was obtained as a result of homomorphic operations. The operation history tells exactly how the particular ChooserPoly was obtained from freshly encrypted ciphertexts. Those ChooserPolys that model freshly encrypted ciphertexts have their operation history set to a special fresh value, with no inputs. We use ChooserPolys with NULL operation history to model plaintext polynomials instead of ciphertexts. Whenever a homomorphic operation is performed on one or more ChooserPolys, a new one is created with (CP1) and (CP2) computed from (CP1) and (CP2) of the inputs. Finally, the operation histories of the inputs are fully cloned and merged to create (CP3).

We still have not mentioned the parameter w, or the decomposition bit count $\log_2 w$. For efficiency reasons we prefer to have $\log_2 w$ be an integer in the range $\{1,\ldots,\lceil \log_2 q \rceil\}$, i.e. we always take w to be a power of 2. It is clear from the estimates in IV-A that a larger w corresponds to larger noise growth in homomorphic multiplication, but it also makes the operation faster. Thus, the procedure for selecting w is as follows:

(1) Start with $\log_2 w = \lceil \log_2 q \rceil$.

(2) Use the inherent noise growth simulator to find out whether decryption can be expected to work with encryption parameters (n,q,t,σ,w).

(3) If decryption can not be expected to succeed, decrease $\log_2 w$, and go to (2), unless $\log_2 w$ is already too small to be efficient (bound can be determined by user), in which case increase (n,q) and go to (1).

Finally, we need to explain in more detail how the pairs (n,q) are chosen. For security and efficiency reasons, we always take n be a power of 2, and choose q from a hard-coded list of prime numbers of a certain form, whose sizes were presented above in Table IV. These choices

are estimated to yield a security level of well over 128 bits (see [20]), and are certainly a conservative choice[3].

Our automatic parameter selection module contains also tools analogous to BalancedEncoder, Encryptor, and Evaluator. This makes it very easy for the user to switch from running their code on ChooserPoly objects to running it on actual data. More precisely, the following classes are provided:

ChooserPoly:
Contains information about the approximate size of a plaintext polynomial, and an *operation history*, as was explained above.

ChooserEncoder:
This class can be used analogously to BalancedEncoder. If the user knows beforehand some numbers going into the computation, e.g. coefficients of a linear function to be evaluated on the encrypted data, they can be converted into ChooserPoly objects with NULL operation history using the encode function in ChooserEncoder.

ChooserEncryptor:
Most importantly, this class contains a function encrypt that can be used to change the operation history in a ChooserPoly from NULL to fresh. The same result can be achieved by calling the set_fresh member function of the particular ChooserPoly.

ChooserEvaluator:
The ChooserEvaluator class is used to perform operations on the ChooserPoly objects. The operations take a varying number of ChooserPolys as input parameters, and always output a new ChooserPoly with updated values for (CP1)–(CP3).

Once the user has performed all of their computations on the ChooserPoly objects, they can use the function select_parameters of ChooserEvaluator to obtain an optimized set of encryption parameters encapsulated in an instance of EncryptionParameters.

C. Examples

Here we present a few simple examples of using the automatic parameter selection module.

Suppose the user wants to compute $12 \cdot 345 + 6789$ in encrypted form. Consider the following C++ code:

```
ChooserEncoder encoder;
ChooserEncryptor encryptor;
ChooserEvaluator evaluator;

ChooserPoly plain1 = encoder.encode(12);
ChooserPoly plain2 = encoder.encode(345);
ChooserPoly plain3 = encoder.encode(6789);
ChooserPoly enc1 = encryptor.encrypt(plain1);
ChooserPoly enc2 = encryptor.encrypt(plain2);
ChooserPoly enc3 = encryptor.encrypt(plain3);

ChooserPoly prod = evaluator.multiply(enc1, enc2);
ChooserPoly result = evaluator.add(prod12, enc3);

EncryptionParameters parms = evaluator. select_parameters(result);
```

This stores a working set of encryption parameters in parms, which the user can read and use. Once the user knows which encryption parameters they want to use, and have set up the cryptosystem accordingly, it is extremely easy to convert the above code to run on real data. The only change needed is, instead of creating ChooserEncoder, ChooserEncryptor, and ChooserEvaluator, to create an encoder, encryptor, and evaluator as usual, e.g.

```
encryption_parameters (Listing II.1) key_generator (Listing
II.2) encryption_tools (Listing II.3)

BalancedEncoder encoder(parms.plain_modulus());
Encryptor encryptor(parms, public_key);
Evaluator evaluator(parms, evaluation_keys);
```

and change the ChooserPoly objects into BigPoly objects. Then BigPoly result will contain the

encryption of $12 \cdot 345 + 6789 = 10929$.

Typically whoever chooses the parameter sizes does not know exactly what the input data is, but only an estimate of its size. For example, if we know that enc1, enc2, and enc3 will contain encryptions of balanced base-3 encoded numbers, with encodings of length at most 9, we could use

```
ChooserEvaluator evaluator;

ChooserPoly enc1(9,1);
ChooserPoly enc2(9,1);
ChooserPoly enc3(9,1);

ChooserPoly prod = evaluator.multiply(enc1, enc2);
ChooserPoly result = evaluator.add(prod12, enc3);

EncryptionParameters parms = evaluator.
select_parameters(result);
```

[3] An expert user might benefit from switching to slightly less secure parameters (larger q, smaller n), and this is certainly easy to do by changing the values in the code, but is also highly discouraged without a very good understanding of the security results.

to obtain appropriate encryption parameters. The constructor of ChooserPoly takes (CP1) and (CP2) as input parameters. The operation histories of the three ChooserPolys are set by default to fresh.

Next we demonstrate choosing parameters for computing a linear combination of encrypted numbers with public coefficients. Suppose we know that all of the encrypted numbers are encoded using balanced base-7 encoding, and have length at most 20 terms[5]. Suppose that the coefficients are stored as integers in an std::vector<int>. The following C++ code can be used to find an appropriate set of encryption parameters:

```
#include <vector> …

ChooserEncoder encoder(7);
ChooserEncryptor encryptor; ChooserEvaluator evaluator;

std::vector<int> c_ints {/* list of coeffs */}; int c_count = c_ints.size();

std::vector<ChooserPoly> c_cps; for(int i=0;
i<c_count; ++i)
{ c_cps.push_back(encoder.encode(c_ints[i]));
}

std::vector<ChooserPoly> encs(
c_count, ChooserPoly(20, 3));

ChooserPoly result = evaluator.multiply_plain( encs[0], c_cps[0]);
for (int i = 1; i < c_count; ++i)
{
ChooserPoly term = evaluator.multiply_plain( encs[i], c_cps[i]);
result = evaluator.add(result, term);
}

EncryptionParameters parms = evaluator. select_parameters(result);
```

V. Biomedical Applications

A. Sample tasks

There are many different types of analyses which researchers or healthcare professionals may wish to perform on sensitive genomic or medical data. For example, Genome-Wide Association Studies (GWAS) typically perform statistical calculations across a large population, such as computing Minor Allele Frequencies (MAFs), χ^2-statistics, Pearson goodness-of-fit tests, tests for association between different loci in the genome, estimates of haplotype frequencies, and tests for association of a genotype with a disease. These, and other statistical analysis tools, are available and widely used in the R Project [27].

[5] So the numbers have absolute value at most $(7^{20}-1)/2$.

Based on earlier internal implementations of homomorphic encryption which were hand-tuned, not publicly available, and not very flexible, performance numbers for many such computations on homomorphically encrypted genomic data were reported in [18], [19], [2]. Most recently, [18] implemented MAFs, χ^2-statistics, Hamming distance, and edit distance for sequence matching, which were the tasks in the Secure Genome Analysis Contest run by iDASH, and funded by NIH [17]. The implementation in [19] was written using the Magma software package [3], and demonstrated encodings and performance numbers for many functions from the R package: the Pearson goodness-of-git and χ^2-tests to test for deviation from Hardy-Weinberg equilibrium, various measures of linkage disequilibrium to test for association in the genotypes at two different loci in a genome, the Estimation Maximization (EM) Algorithm to estimate haplotype frequencies from genotype counts, and the Cochran-Armitage Test for Trend (CATT) to determine if a candidate allele is associated with a disease.

In [2], logistic regression and the Cox proportional hazard model were implemented as representative examples for disease prediction. A private cloud service for predicting cardiovascular disease (CVD) was demonstrated on homomorphically encrypted data using a model based on logistic regression, and shown at the AAAS Meeting 2014 Newsroom. To apply logistic regression to homomorphically encrypted data, we use a polynomial approximation to the function which approximates the prediction well enough in a certain range. Logistic regression has been commonly used to predict whether a patient will survive or suffer from various diseases, including cardiovascular disease (CVD), diabetes, probability of survival in blunt trauma, testing gender as a predictor of mortality after heart surgery, correlating genotypes with the risk of cardiovascular disease, and relating protein abnormalities with occurrence of diabetes [13].

B. Practical considerations

The statistical functions mentioned above often take inputs which are integers or real numbers. For example, frequency counts for MAF and haplotype frequencies are represented as integers, and health data input to predictive models using logistical regression are often real numbers. The encoding methods described in III-A and III-B can be used to significantly improve the

parameters and performance of homomorphic encryption for such applications. On the other hand, tasks like sequence matching often take discrete inputs, such as strings of genomic data. In such cases the best option might be to use bit-wise encryption of inputs, and use the CRT techniques to pack multiple bits in one plaintext/ciphertext pair, as was briefly described in III-D. In this section we demonstrate how to concretely use SEAL for tasks such as these.

As our first example we discuss using the logistic regression model for predicting the likelihood of a patient developing diabetes [2]. A predictive equation to screen for diabetes was developed based on logistic regression in [31]. The equation was computed from data on more than 1,000 Egyptian patients with no history of diabetes. The predictive variables used were: age (a), sex, BMI, number of hours since the last food or drink (PT: postprandial time), and Random Capillary Plasma Glucose level (RCPG). The study was cross-validated on a sample of more than 1,000 American patients. The predictive equation calculated is

$$P(x) := \frac{e^x}{e^x + 1},$$

with the following logistic regression parameters: x

$$= -10.0382 + 0.0331 \cdot \mathrm{a}$$
$$+ 0.0308 \cdot \mathrm{RCPG} + 0.2500 \cdot \mathrm{PT}$$
$$+ 0.5620 \cdot (\text{if female}) + 0.0346 \cdot \mathrm{BMI},$$

where age is given in years, random plasma glucose (RPG) in mg/dl, and postprandial time (PT) in hours. Undiagnosed diabetes is predicted if the value is greater than 0.20 (20%). Thus only one digit of accuracy is required beyond the decimal point when computing the value of the predictive function approximately.

The sigmoid function $P(x)$ can be approximated near $x = 0$ by the Taylor series

$$P(x) = \frac{1}{2} + \frac{1}{4}x - \frac{1}{48}x^3 + \frac{1}{480}x^5 - \frac{17}{80640}x^7 + O(x^9),$$

which we can attempt to evaluate homomorphically on our input data.

First set up the scheme and the encoder, and encrypt some sample patient data:

```
#include <vector> ...

encryption_parameters (Listing II.1) key_generator (Listing
II.2) encryption_tools (Listing II.3)

BalancedFractionalEncoder encoder( parms.plain_modulus(),
  parms.poly_modulus(),
256, 16);

BigPoly a = encoder.encode(42);
BigPoly RCPG = encoder.encode(115); BigPoly PT =
encoder.encode(8);
BigPoly female = encoder.encode(1); BigPoly BMI =
encoder.encode(20.2); std::vector<BigPoly> data {
encryptor.encrypt(a), encryptor.encrypt(RCPG),
encryptor.encrypt(PT), encryptor.encrypt(female),
encryptor.encrypt(BMI)
};
```

Next, encode the coefficients needed to evaluate x:

```
std::vector<BigPoly> xweights { encoder.encode(0.0331),
  encoder.encode(0.0308), encoder.encode(0.2500),
  encoder.encode(0.5620), encoder.encode(0.0346)
};
BigPoly translate = encoder.encode(-10.0382);
```

Computing x is now easy using the multiply_plain and add functions:

```
BigPoly x = evaluator.multiply_plain(data[0], xweights[0]);
for (int i = 1; i<5; ++i)
{
BigPoly prod = evaluator.multiply_plain( encrypted_data[i],
  encoded_xweights[i]);
x = evaluator.add(x, prod);
}
x = evaluator.add_plain(x, translate);
```

Now we come to evaluating the Taylor expansion. First encode the coefficients of the expansion:

```
std::vector<BigPoly> taylor_coeffs {
encoder.encode(1.0 / 4), encoder.encode(-1.0 / 48),
encoder.encode(1.0 / 480), encoder.encode(-17.0 / 80640)
};
BigPoly taylor_constant = encoder.encode(1.0 / 2);
```

To compute the odd powers of x we can use the member function exponentiate of Evaluator. Computing the Taylor expansion is now easy:

```
BigPoly result = evaluator.multiply_plain( x, taylor_coeffs[0]);
for (int i = 1; i<4; ++i)
{
BigPoly power_of_x = evaluator.exponentiate( x, 2 * i + 1);
BigPoly prod = evaluator.multiply_plain( power_of_x, taylor_coeffs[i]);
result = evaluator.add(result, prod);
} result = evaluator.add_plain(result, taylor_constant);
```

Finally, result can be decrypted and decoded:

```
double plain_result = encoder.decode(decryptor. decrypt(result));
```

The computation above is, however, not a very optimal solution. In fact, the parameters given in Listing II.1 are not nearly large enough to deal with this problem. Everything works well until the sigmoid function has to be evaluated. Problems arise when real numbers encoded with the fractional encoders are raised to high powers, such as 7. The reason is that even if only very few terms of precision are used, the number of cross terms between those few high degree terms in the exponentiation quickly becomes enormous, and dominates the growth of the coefficients of the plaintext polynomial. This forces us to increase t significantly, which in turn forces us to use a much larger (n,q) pair than what is given in Listing II.1. Another unrelated problem is that the sigmoid function is not very well approximated by a Taylor expansion, and the quality of the result in this case depends hugely on the exact value of x.

There are numerous ways to solve these problems using more complicated neural networks to perform the prediction. Such neural networks can be made to use low degree polynomials as activation functions to yield models better suited for homomorphic computations than the above logistic regression model. Another option is to return x instead of returning the troublesome probability $P(x)$.

VI. Examples

In this section we present complete examples for using the SEAL library. Our examples are in C++, but are easy to convert to use the C# wrappers in the SEALNET library. All of the examples we present here use the basic setup presented in Listings II.1, II.2, and II.3.

As the first example, consider the following encrypted computation of $(x^2-1)\cdot(x^3-2x+1)-(x^3+x^2+x+1)$.

```
#include "seal.h" #include
<iostream>

using namespace std; using
namespace seal;

int main()
{ encryption_parameters (Listing II.1) key_generator
   (Listing II.2) encryption_tools (Listing II.3)

/* Note that a negative coefficient y is expressed as t-y */
BigPoly plain1("1xˆ2 + 3FF");
BigPoly plain2("1xˆ3 + 3FExˆ1 + 1");
BigPoly plain3("1xˆ3 + 1xˆ2 + 1xˆ1 + 1");

/* Now encrypt plain1, plain2, plain3 */
BigPoly enc1 = encryptor.encrypt(plain1);
BigPoly enc2 = encryptor.encrypt(plain2);
BigPoly enc3 = encryptor.encrypt(plain3);

/* Use Evaluator to do the computation */
BigPoly prod = evaluator.multiply(enc1, enc2); BigPoly negenc3 =
evaluator.negate(enc3);
BigPoly result = evaluator.add(prod, negenc3);

BigPoly plain_result = decryptor.decrypt( result);

/* Now print the result: xˆ5-4xˆ3+x-2 */ cout << "Result: " <<
plain_result.to_string()
<< endl;

return 0;
}
```

Here is an example of computing the weighted average of 5 real numbers (in the vector numbers), with given public weights (in the vector weights).

```
#include "seal.h"
#include <iostream> #include
<vector>

using namespace std; using
namespace seal;

int main()
{ encryption_parameters (Listing II.1) key_generator
   (Listing II.2) encryption_tools (Listing II.3)

/* We need the fractional encoder */
BalancedFractionalEncoder encoder(parms. plain_modulus(),
   parms.poly_modulus(),
256, 64);

vector<BigPoly> numbers{
encryptor.encrypt(encoder.encode(6.12)),
encryptor.encrypt(encoder.encode(1.10)),
encryptor.encrypt(encoder.encode(8.43)),
encryptor.encrypt(encoder.encode(9.30)),
encryptor.encrypt(encoder.encode(7.05))
};

vector<BigPoly> weights{ encoder.encode(0.20),
   encoder.encode(0.20), encoder.encode(0.35),
   encoder.encode(0.15), encoder.encode(0.20)
};
BigPoly denom = encoder.encode(0.2);

/* Multiply numbers by weights and add them up to result */
BigPoly result = evaluator.multiply_plain( numbers[0], weights[0]);
for (int i = 1; i<5; ++i)
{
BigPoly prod = evaluator.multiply_plain( numbers[i], weights[i]);
result = evaluator.add(result, prod);
}

/* Finally divide by 5 */ result = evaluator.multiply_plain(result,
denom);

/* Now decrypt/decode and print the result:
*/
BigPoly plain_result = decryptor.decrypt( result);
cout << "Result: " << encoder.decode(
plain_result) << endl;

return 0;
}
```

The above code will print the correct answer: 1.4399.

A. Performance

To give a rough idea of the overhead for doing computation on homomorphically encrypted data, we give some sample timings for the SEAL library when running on a 2.00 GHz machine using a single thread. Much better performance can be achieved by parallelizing the computations. These timings were obtained by averaging across 25 runs of the operations.

For a smaller parameter set with $n = 1024$, and coefficient modulus q of size roughly 48 bits, the time required for a homomorphic multiplication of ciphertexts is around 92 milliseconds, including the costly relinearization step II-E. For a larger parameter set with $n = 4096$, and coefficient modulus q of size roughly 127 bits, the time required for a homomorphic multiplication of ciphertexts is around 291 milliseconds, including relinearization.

REFERENCES

[1] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding*, pages 45–64. Springer, 2013.

[2] Joppe W Bos, Kristin Lauter, and Michael Naehrig. Private predictive analysis on encrypted medical data. *Journal of biomedical informatics*, 50:234–243, 2014.

[3] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[4] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[5] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *PublicKey Cryptography–PKC 2013*, pages 1–13. Springer, 2013.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[7] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehle. Classical hardness of learning with errors.´ In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 575–584. ACM, 2013.

[8] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *FOCS*, pages 97–106. IEEE, 2011.

[9] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011*, pages 505–524. Springer, 2011.

[10] Zvika Brakerski and Vinod Vaikuntanathan. Lattice-based FHE as secure as PKE. In Moni Naor, editor, *ITCS*, pages 1–12. ACM, 2014.

[11] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. Cryptology ePrint Archive, Report 2015/132, 2015. http://eprint.iacr.org/.

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

[13] Andreas Festa, Ken Williams, Anthony J.G. Hanley, James D. Otvos, David C. Goff, Lynne E. Wagenknecht, and Steven M. Haffner. Nuclear magnetic resonance lipoprotein abnormalities

in prediabetic subjects in the insulin resistance atherosclerosis study. *Circulation*, 111(25):3465–3472, 2005.

[14] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[15] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology– CRYPTO 2012*, pages 850–867. Springer, 2012.

[16] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[17] iDASH Privacy & security workshop 2015. Secure genome analysis competition. http://www.humangenomeprivacy.org/2015/competition-tasks.html.

[18] Miran Kim and Kristin Lauter. Private genome analysis through homomorphic encryption. Cryptology ePrint Archive, Report 2015/965, 2015. http://eprint.iacr.org/.

[19] Kristin Lauter, Adriana Lopez-Alt, and Michael Naehrig. Pri-´ vate computation on encrypted genomic data. In *Progress in Cryptology-LATINCRYPT 2014*, pages 3–27. Springer, 2014.

[20] Tancrede Lepoint and Michael Naehrig. A comparison of the` homomorphic encryption schemes fv and yashe. In *Progress in Cryptology–AFRICACRYPT 2014*, pages 318–335. Springer, 2014.

[21] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Topics in Cryptology– CT-RSA 2011*, pages 319–339. Springer, 2011.

[22] Adriana Lopez-Alt, Eran Tromer, and Vinod Vaikuntanathan.´ On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the fortyfourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

[23] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):43, 2013.

[24] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *Advances in Cryptology–EUROCRYPT 2012*, pages 700–718. Springer, 2012.

[25] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[26] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2009.

[27] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[28] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[29] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[30] Damien Stehle and Ron Steinfeld.´ Making ntru as secure as worst-case problems over ideal lattices. In *Advances in Cryptology–EUROCRYPT 2011*, pages 27–47. Springer, 2011.

[31] Bahman P Tabaei and William H Herman. A multivariate logistic regression equation to screen for diabetes development and validation. *Diabetes Care*, 25(11):1999–2003, 2002.

[32] Joop van de Pol and Nigel P Smart. Estimating key sizes for high dimensional lattice-based systems. In *Cryptography and Coding*, pages 290–303. Springer, 2013.

[33] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.