Simple Encrypted Arithmetic Library - SEAL v2.1

Hao Chen[1], Kim Laine[2], and Rachel Player[3]

[1] Microsoft Research, USA
haoche@microsoft.com
[2] Microsoft Research, USA
kim.laine@microsoft.com
[3] Royal Holloway, University of London, UK**
rachel.player.2013@live.rhul.ac.uk

1 Introduction

Traditional encryption schemes, both symmetric and asymmetric, were not designed to respect the algebraic structure of the plaintext and ciphertext spaces. Many schemes, such as ElGamal (resp. e.g. Paillier), are multiplicatively homomorphic (resp. additively homomorphic), so that one can perform certain limited types of computations directly on the encrypted data and have them pass through the encryption to the underlying plaintext data, without requiring access to any secret key(s). The restriction to a one particular type of operation is very strong, however, and instead a much more powerful *fully* homomorphic encryption scheme, that respects two algebraic operations between the plaintext and ciphertext spaces, would be needed for many interesting applications. The first such encryption scheme was presented by Craig Gentry in his famous work [16], and since then researchers have introduced a number of new and more efficient fully homomorphic encryption schemes.

Despite the promising theoretical power of homomorphic encryption, the practical side still remains somewhat underdeveloped. Recently new implementations, new data encoding techniques, and new applications have started to improve the situation, but much remains to be done. In 2015 we released the *Simple Encrypted Arithmetic Library - SEAL* with the goal of providing a well engineered and documented homomorphic encryption library, with no external dependencies, that would be easy to use both by experts and by non-experts with little or no cryptographic background. The library is available at `http://sealcrypto.codeplex.com`, and is licensed under the MSR License Agreement.

Recently a large number of major changes were implemented in SEAL, and the new version was released as SEAL v2.0. This involved major changes in the public API, and a change of the underlying encryption scheme. Since the release of SEAL v2.0 several improvements have been made. Most of these are very important performance-related improvements, but there are also a few changes and additions to the public API. These improvements have now been released as SEAL v2.1. In this document we describe in detail this new release, and hope to provide a practical guide to using homomorphic encryption for a wide audience. The reader is also advised to go over the code examples that come with the library, and to read through the detailed comments. For users of previous versions of SEAL (both v2.0 and earlier) we hope to provide clear instructions for how to port old code to use SEAL v2.1. An introductory paper to an older version of SEAL was given in [12], which the user new to SEAL v2.1 may also find helpful as large parts of the API have remained unchanged. This document is an extension of a similar document that accompanied the release of SEAL v2.0 [19].

** Much of this work was done during an internship at Microsoft Research, Redmond.

1.1 Roadmap

Section 1.2 is directly taken from [19] and gives an overview of the changes for SEAL v2.0 compared to versions prior to it. In Section 1.3 we give an overview of changes moving from SEAL v2.0 to SEAL v2.1, which are expanded upon in the other sections of this document. In Section 2 we define notation and parameters that are used throughout this document. In Section 3 we give the description of the Fan-Vercauteren homomorphic encryption scheme (FV) – as originally specified in [15] – and in Section 4 we describe how SEAL differs from this original description. In Section 5 we discuss the expected noise growth behavior of SEAL ciphertexts as homomorphic evaluations are performed. In Section 6 we discuss the available ways of encoding data into SEAL plaintexts. In Section 7 we discuss the selection of parameters for performance, and describe the automatic parameter selection module. In Section 8 we discuss the security properties of SEAL.

1.2 Overview of Changes in SEAL v2.0

In this subsection we highlight some of the changes in SEAL v2.0 compared to previous versions. In addition to the changes discussed below a huge number of bugs have been fixed, and some core functions have been optimized.

Remark 1. Whenever we refer to (either implicitly or explicitly) an implementation of YASHE', we mean the implementation in the versions of SEAL prior to SEAL v2.0. Note that YASHE' is no longer available in SEAL v2.0, and the only cryptosystem implemented is now FV.

Remark 2. Whenever we refer to (either implicitly or explicitly) implementations of *encryptor*, *decryptor*, *key generator*, *encryption parameters*, *coefficient modulus*, *plaintext modulus*, etc., we mean classes, objects, or variables with corresponding names in SEAL (`Encryptor`, `Decryptor`, `KeyGenerator`, `EncryptionParameters`, `coeff_modulus`, `plain_modulus`, etc.). We use *unsigned integers*, *polynomials*, and *polynomial arrays* to refer to the SEAL objects `BigUInt`, `BigPoly`, and `BigPolyArray`.

New encryption scheme Previous versions of SEAL used the scheme YASHE', introduced by Bos, Lauter, Loftus, and Naehrig in [3], as the underlying encryption scheme. SEAL v2.0 uses the Fan-Vercauteren scheme, which we will refer to as the *FV scheme*, introduced by Fan and Vercauteren in [15] (see also Section 3). This change improves both security and performance. In particular, FV is more secure because it relies only on the RLWE assumption (see Section 8). It also has better ciphertext noise growth properties (see Section 5) roughly due to a smaller size secret key (represented by a `BigPoly` object in SEAL v2.0). As a result, in many cases it is now possible to use smaller parameters (`poly_modulus` and `coeff_modulus`) than before, resulting in significantly improved performance.

While plaintext elements remain the same as before, i.e. represented by `BigPoly` objects with coefficients reduced modulo `plain_modulus`, in the FV scheme a freshly encrypted ciphertext is an array of two polynomials, represented by instances of a new class `BigPolyArray`. In both schemes the secret key is a `BigPoly`. In FV the public key is a `BigPolyArray` of size 2, whereas in YASHE' it was represented by a single `BigPoly`.

Relinearization does not occur by default To obtain certain cryptographic properties (compactness, circuit privacy), textbook-FV as described in [15] performs a *relinearization*

operation after every homomorphic multiplication. The reason is that homomorphic multiplication in fact increases the size of the output ciphertext `BigPolyArray`. Precisely, the result of multiplying two ciphertexts of sizes M and N results in a ciphertext of size $M + N - 1$. Relinearization can be used to reduce the size down from 3 to 2 after every multiplication, preventing the ciphertext size from leaking information about the evaluated arithmetic circuit. In textbook-YASHE' homomorphic multiplication also involves the production of an intermediate ciphertext, which should be relinearized to produce the final output ciphertext [3]. In previous versions of SEAL the function `Evaluator::multiply` returned this output ciphertext. In SEAL v2.0 we do not perform relinearization by default anymore, so instead `Evaluator::multiply` returns a ciphertext of size $M + N - 1$ (given inputs of size M and N). If desired, subsequent relinearization may be done using `Evaluator::relinearize`. The reason for this is that while in many cases relinearizing after every multiplication is a good strategy, it is not optimal, and in some cases the user might be able to squeeze out more performance by deferring relinearization until a later point, and instead work temporarily with larger ciphertexts. We also extend the idea of relinearization in SEAL v2.0 to reducing a ciphertext of arbitrary size down to any size at least 2 (by default to size 2). See Section 4 for further discussion on the generalization of multiplication and relinearization.

Another reason for not relinearizing by default is that the performance of `Evaluator::relinearize` depends strongly on the choice of the parameter `decomposition_bit_count` in `EncryptionParameters`. A reasonable choice for the decomposition bit count is between 1/10 and 1/2 of the significant bit count of the coefficient modulus, but since it affects both ciphertext noise growth and performance, it is hard to determine the optimal choice without knowing the details of the particular computation. On the other hand, the choice of the decomposition bit count does not matter if there will be no relinearization in the computation, and since relinearization does not occur by default anymore, the constructor of `EncryptionParameters` can set it automatically to 0 (signaling that no relinearization will be performed). This frees the user from having to worry about `decomposition_bit_count` unless they choose to.

To be able to relinearize a ciphertext, the owner of the secret key must have generated enough evaluation keys that need to be subsequently given as input to the constructor of `Evaluator`. More precisely, if a ciphertext has size equal to K, then $K-2$ evaluation keys will be needed to relinearize it down to any size less than K. To generate k evaluation keys with the key generator, the owner of the secret key can call `KeyGenerator::generate` with the parameter k. Of course, if the key generator is instantiated with a decomposition bit count of 0 (see the above paragraph), the `generate` function can only be called with parameter $k = 0$ (the default value). Previously the constructor of `Evaluator` always required both encryption parameters and evaluation keys, but now if no evaluation keys have been generated the evaluator can be constructed by only passing it a set of encryption parameters. If the evaluator is constructed in such a way, it will not be possible to use `Evaluator::relinearize`. To conclude, in order to perform relinearization, the user must first set the decomposition bit count to a non-zero value, then generate an appropriate number of evaluation keys with `KeyGenerator::generate`, and finally pass the generated `EvaluationKeys` instance to the constructor of `Evaluator`.

Validity of ciphertexts In SEAL v2.0 we adopt the philosophy that for a ciphertext to be valid it should not reveal any information about the underlying plaintext beyond the fact that it is the result of the evaluation of an arithmetic circuit with properties that might be inferred from the noise level and the size of the ciphertext. For this reason the functions

`Evaluator::multiply_plain` with a plaintext multiplier 0, and `Evaluator:exponentiate` with exponent 0, are now not allowed, and will result in SEAL throwing an exception.

Thread safety Earlier versions of SEAL used a memory pool (class `util::MemoryPool`), but this was not thread-safe. As a result, each `Encryptor`, `Decryptor`, `Evaluator`, and all the encoders had their own memory pools, and in a multi-threaded application each thread would have to contain its own local instance of these objects if they were to be used concurrently. The `util::MemoryPool` class is now thread-safe, and there is one single global instance of if used by the entire library.[4] Consequently, a single instance of each of the classes mentioned above can now be used concurrently from any number of threads. As before, classes such as `BigUInt`, `BigPoly`, `BigPolyArray`, and `PolyCRTBuilder` are not fully thread-safe, and the limitations are described in the comments in the header files.

Default source of randomness Previous versions of SEAL assumed that the user supplied their own cryptographic source of randomness to `EncryptionParameters`, and if this was not done it did not use real randomness (for testing purposes). As supplying a custom randomness source is slightly non-trivial, the default behavior was changed to use `std::random_device` for randomness. For more details, see Section 7.

1.3 From SEAL v2.0 to SEAL v2.1

The changes from SEAL v2.0 to SEAL v2.1 are mostly technical improvements, and involve only a few minor changes or additions to the public API. However, here we will focus on describing the API changes.

Inherent noise The earlier `inherent_noise` functions have been removed from SEAL v2.1, and replaced by a member function `Decryptor::inherent_noise`. As typically the user never cares about the exact `BigUInt` value of the inherent noise but only its bit length, a convenient function `Decryptor::inherent_noise_bits` was created to instead return the significant bit count of the inherent noise. Similarly, the function `inherent_noise_max` has been removed from SEAL v2.1, and replaced by a member function `EncryptionParameters ::inherent_noise_max`. As the new `inherent_noise` function, `inherent_noise_max` also comes with a variant `inherent_noise_bits_max` that instead returns the significant bit count of the maximum noise.

Simpler encoders The `BinaryEncoder` and `BalancedEncoder` have been merged into one class called `IntegerEncoder`, which takes any integer base ≥ 2 as an argument, and under the hood works as either a `BinaryEncoder` or as a `BalancedEncoder`, depending on the base. Moreover, note that `BalancedEncoder` (and hence `IntegerEncoder`) supports now also even integers as base. Similar changes were made to merge `BinaryFractionalEncoder` and `BalancedFractionalEncoder` into one `FractionalEncoder` class. Both `IntegerEncoder` and `FractionalEncoder` use base 2 as a default when no base is explicitly given.

[4] The global memory pool can be accessed through `util::MemoryPool::default_pool()`.

`exponentiate` and `multiply_many` relinearize by default In SEAL v2.1 we have decided to enable automatic relinearization for the two functions `Evaluator:exponentiate` and `Evaluator::multiply_many`. Relinearization to size 2 is performed after each multiplication, and the final result of the computation will always be a ciphertext of size 2. This is in contrast to `Evaluator::multiply`, where no relinearization is done. Note that the user is responsible for ensuring that enough evaluation keys have been generated and given to the constructor of `Evaluator`.

Faster operations with NTT In order to use the number theoretic transform (NTT) to speed up some essential operations – encryption, relinearization and decryption – we require `coeff_modulus` to be a prime q with the property that $q \equiv 1 \pmod{2n}$, where n is the degree of `poly_modulus`. We use David Harvey's algorithm for NTT as described in [18], which additionally requires that $4q \leq \beta$, where

$$\beta = 2^{64\lceil \log(q)/64 \rceil}$$

denotes the *word size*. Thus, NTT will only be used if $2n|(q-1)$ and $4q \leq \beta$ holds. Otherwise slower algorithms will be used.

New default parameters In SEAL v2.1 we have updated the set of default parameters based on more recent security estimates (see [1]). Our default parameters are chosen conservatively to ensure security against improvements to known attacks. See Table 3 for the new list of default parameters.

Changes to `PolyCRTBuilder` The `PolyCRTBuilder` class has changed significantly. The constructor takes an `EncryptionParameters` instance as input, and checks that the parameters are valid and appropriate for use with batching functionality. In particular, it verifies that $2n|(t-1)$, where t is the `plain_modulus`. For manipulating slot contents, the `PolyCRTBuilder` class now contains only the `compose` and `decompose` functions, whereas SEAL v2.0 had in addition functions such as `get_slot` and `set_slot`. However, `compose` and `decompose` are now hugely faster than before due to the use of NTT. More precisely, they are implemented now via a *negacyclic* variant of the Number Theoretic Transform (see e.g. [21]).

Encryption parameter qualifiers Depending on what encryption parameters the user chooses, different functionalities are enabled in the library. We call these different properties of the encryption parameters their *qualifiers*. Given an instance of `EncryptionParameters`, the user can call `EncryptionParameters::get_qualifiers` to return the set of qualifiers as an instance of a struct `EncryptionParameterQualifiers`, which is a simple container for several Boolean flags that tell which features will be enabled with the particular parameter set. If the set of parameters is not valid for use with SEAL v2.1, the flag `parameters_set` will be `false`. It is impossible to create tools such as `Encryptor` or `Decryptor` with invalid parameters.

There is one important change in what parameters are considered valid. We no longer allow `poly_modulus` to be anything but a power-of-2 cyclotomic polynomial, i.e. of the form $x^n + 1$, where n is a power of 2.

Faster squaring We have added a function `Evaluator::square` which uses a different algorithm for homomorphic multiplication of a ciphertext with itself. This is significantly faster than using `Evaluator::multiply`. The classes `SimulationEvaluator` and `ChooserEvaluator` contain a similar `square` function.

Self-assignment in `Evaluator` Earlier many of the functions in `Evaluator` failed when the destination was equal to one of the inputs. This problem has now been corrected.

No throwing on decoder overflow Earlier versions of SEAL threw an exception when an encoder failed to decode a plaintext polynomial due to the output not fitting into the output container. This is a potential security issue if the user does not catch the exception and deal with it silently, as any passively observing party might notice the program crashing and be able to deduce information about the underlying plaintext. Typically the user might want to catch such an exception and deal with it accordingly, but since SEAL v2.1 does not yet have its own exception classes, for now throwing on decoder overflow is disabled.

2 Notation

We use $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$, and $\lfloor \cdot \rceil$ to denote rounding down, up, and to the nearest integer, respectively. When these operations are applied to a polynomial, we mean performing the corresponding opearation to each coefficient separately. The norm $\| \cdot \|$ always denotes the infinity norm. We denote the reduction of an integer modulo t by $[\cdot]_t$. This operation can also be applied to polynomials, in which case it is applied to every integer coefficient separately. The reductions are always done into the symmetric interval $[-t/2, t/2)$. $\log_a$ denotes the base-a logarithm, and log always denotes the base-2 logarithm.Table 1 below lists commonly used parameters, and in some cases their corresponding names in SEAL v2.1.

3 The FV Scheme

In this section we give the definition of the FV scheme as presented in [15].

3.1 Plaintext Space and Encodings

In FV the plaintext space is $R_t = \mathbb{Z}_t[x]/(x^n + 1)$, that is, polynomials of degree less than n with coefficients modulo t. We will also use the ring structure in R_t, so that e.g. a product of two plaintext polynomials becomes the product of the polynomials with x^n being converted to a -1. The homomorphic addition and multiplication operations on ciphertexts (that will be described later) will carry through the encryption to addition and multiplications operations in R_t.

If one wishes to encrypt (for example) an integer or a rational number, it needs to be first encoded into a plaintext polynomial in R_t, and can be encrypted only after that. In order to be able to compute additions and multiplications on e.g. integers in encrypted form, the encoding must be such that addition and multiplication of encoded polynomials in R_t carry over correctly to the integers when the result is decoded. SEAL provides a few different encoders for the user's convenience. These are discussed in more detail in Section 6 and demonstrated in the SEALExamples project that comes with the code.

Parameter	Description	Name in SEAL (if applicable)
q	Modulus in the ciphertext space (coefficient modulus)	`coeff_modulus`
t	Modulus in the plaintext space (plaintext modulus)	`plain_modulus`
n	A power of 2	
$x^n + 1$	The polynomial modulus which specifies the ring R	`poly_modulus`
R	The ring $\mathbb{Z}[x]/(x^n + 1)$	
R_a	The ring $\mathbb{Z}_a[x]/(x^n + 1)$, i.e. same as the ring R but with coefficients reduced modulo a	
w	A base into which ciphertext elements are decomposed during relinearization	
$\log w$		`decomposition_bit_count`
ℓ	There are $\ell + 1 = \lfloor \log_w q \rfloor + 1$ elements in each component of each evaluation key	
δ	Expansion factor in the ring R ($\delta \leq n$)	
Δ	Quotient on division of q by t, or $\lfloor q/t \rfloor$	
$r_t(q)$	Remainder on division of q by t, i.e. $q = \Delta t + r_t(q)$, where $0 \leq r_t(q) < t$	
χ	Error distribution (a truncated discrete Gaussian distribution)	
σ	Standard deviation of χ	`noise_standard_deviation`
B	Bound on the distribution χ	`noise_max_deviation`

Table 1: Notation used throughout this document.

3.2 Ciphertext Space

Ciphertexts in FV are arrays of polynomials in R_q. These arrays contain at least two polynomials, but grow in size in homomorphic multiplication operations unless relinearization is performed. Homomorphic additions are performed by computing a component-wise sum of these arrays; homomorphic multiplications are slightly more complicated and will be described below.

3.3 Description of Textbook-FV

Let λ be the security parameter. Let w be a base, and let $\ell+1 = \lfloor \log_w q \rfloor + 1$ denote the number of terms in the decomposition into base w of an integer in base q. We will also decompose polynomials in R_q into base-w components coefficient-wise, resulting in $\ell+1$ polynomials. By $a \xleftarrow{\$} \mathcal{S}$ we denote that a is sampled uniformly from the finite set $\mathcal{S}$.

The scheme FV contains the algorithms `SecretKeyGen`, `PublicKeyGen`, `EvaluationKeyGen`, `Encrypt`, `Decrypt`, `Add`, and `Multiply`. These algorithms are described below.

- $\texttt{SecretKeyGen}(\lambda)$: Sample $s \xleftarrow{\$} R_2$ and output $\texttt{sk} = s$.
- $\texttt{PublicKeyGen}(\texttt{sk})$: Set $s = \texttt{sk}$, sample $a \xleftarrow{\$} R_q$, and $e \leftarrow \chi$. Output $\texttt{pk} = ([-(as+e)]_q, a)$.
- $\texttt{EvaluationKeyGen}(\texttt{sk}, w)$: for $i \in \{0, \ldots, \ell\}$, sample $a_i \xleftarrow{\$} R_q$, $e_i \leftarrow \chi$. Output
$$\texttt{evk} = \left([-(a_i s + e_i) + w^i s^2]_q, a_i\right) .$$
- $\texttt{Encrypt}(\texttt{pk}, m)$: For $m \in R_t$, let $\texttt{pk} = (p_0, p_1)$. Sample $u \xleftarrow{\$} R_2$, and $e_1, e_2 \leftarrow \chi$. Compute
$$\texttt{ct} = ([\Delta m + p_0 u + e_1]_q, [p_1 u + e_2]_q) .$$
- $\texttt{Decrypt}(\texttt{sk}, \texttt{ct})$: Set $s = \texttt{sk}$, $c_0 = \texttt{ct}[0]$, and $c_1 = \texttt{ct}[1]$. Output
$$\left[\left\lfloor \frac{t}{q}[c_0 + c_1 s]_q \right\rceil\right]_t .$$
- $\texttt{Add}(\texttt{ct}_0, \texttt{ct}_1)$: Output $(\texttt{ct}_0[0] + \texttt{ct}_1[0], \texttt{ct}_0[1] + \texttt{ct}_1[1])$.
- $\texttt{Multiply}(\texttt{ct}_0, \texttt{ct}_1)$: Compute
$$c_0 = \left[\left\lfloor \frac{t}{q} \texttt{ct}_0[0]\texttt{ct}_1[0] \right\rceil\right]_q ,$$
$$c_1 = \left[\left\lfloor \frac{t}{q} \left(\texttt{ct}_0[0]\texttt{ct}_1[1] + \texttt{ct}_0[1]\texttt{ct}_1[0]\right) \right\rceil\right]_q ,$$
$$c_2 = \left[\left\lfloor \frac{t}{q} \texttt{ct}_0[1]\texttt{ct}_1[1] \right\rceil\right]_q .$$
Express c_2 in base w as $c_2 = \sum_{i=0}^{\ell} c_2^{(i)} w^i$. Set
$$c_0' = c_0 + \sum_{i=0}^{\ell} \texttt{evk}[i][0] c_2^{(i)} ,$$
$$c_1' = c_1 + \sum_{i=0}^{\ell} \texttt{evk}[i][1] c_2^{(i)} ,$$
and output (c_0', c_1').

4 How SEAL Differs from Textbook-FV

In practice, some operations in SEAL are done slightly differently, or in slightly more generality, than in textbook-FV (see Section 3.3). In this section we discuss these differences in detail.

To make clear the generalization of FV operations it is convenient to think of each ciphertext component as corresponding to a particular power of the secret key s. In particular, in a ciphertext $\mathtt{ct} = (c_0, c_1, \ldots c_k)$ of size $k+1$, the c_0 term is associated with s^0, the c_1 term with s^1, and so on, so that the c_k term is associated with s^k.

4.1 Decryption

A SEAL v2.1 ciphertext $\mathtt{ct} = (c_0, \ldots, c_k)$ is decrypted by computing

$$\left[\left\lfloor \frac{t}{q}[\mathtt{ct}(s)]_q \right\rceil\right]_t = \left[\left\lfloor \frac{t}{q}\left[c_0 + \cdots + c_k s^k\right]_q \right\rceil\right]_t .$$

This generalization of decryption (compare to Section 3.3) is handled automatically. The decryption function determines the size of the input ciphertext, and generates the appropriate powers of the secret key which are required to decrypt it. Note that because we consider well-formed ciphertexts of arbitrary length valid, we automatically lose the compactness property of homomorphic encryption. Roughly speaking, compactness states that the decryption circuit should not depend on ciphertexts, or on the function being evaluated. For more details, see [2].

4.2 Multiplication

Consider the `Multiply` function as described in Section 3. The first step that outputs the intermediate ciphertext (c_0, c_1, c_2) defines a function `Evaluator::multiply`[5], and causes the ciphertext to grow in size. The second step defines a function that we call relinearization, implemented as `Evaluator::relinearize`, which takes a ciphertext of size 3 and an evaluation key, and produces a ciphertext of size 2, encrypting the same underlying plaintext. Note that the ciphertext (c_0, c_1, c_2) can already be decrypted to give the product of the underlying plaintexts (see Section 4.1), so that in fact the relinearization step is not necessary for correctness of homomorphic multiplication.

It is possible to repeatedly use a generalized version of the first step of `Multiply` to produce even larger ciphertexts if the user has a reason to further avoid relinearization. In particular, let $\mathtt{ct}_1 = (c_0, c_1, \ldots, c_j)$ and $\mathtt{ct}_2 = (d_0, d_1, \ldots, d_k)$ be two SEAL v2.1 ciphertexts of sizes $j+1$ and $k+1$, respectively. Let the ciphertext output by $\mathtt{Multiply}(\mathtt{ct}_1, \mathtt{ct}_2)$, which is of size $j+k+1$, be denoted $\mathtt{ct}_{\text{mult}} = (C_0, C_1, \ldots, C_{j+k})$. The polynomials $C_m \in R_q$ are computed as

$$C_m = \left[\left\lfloor \frac{t}{q}\left(\sum_{r+s=m} c_r d_s\right) \right\rceil\right]_q .$$

In SEAL v2.1 we define the function `Multiply` (or rather family of functions) to mean this generalization of the first step of multiplication. It is implemented as `Evaluator::multiply`.

[5] This is not quite true, because previous versions of SEAL used YASHE', where ciphertexts did not grow in size, but instead the resulting ciphertext had to instead be decrypted under a single higher power of the secret key. Note that in SEAL v2.1 decryption will require the entire sequence s^0, s^1, s^2.

4.3 Relinearization

The goal of relinearization is to decrease the size of the ciphertext back to (at least) 2 after it has been increased by multiplications as was described in Section 4.2. In other words, given a size $k+1$ ciphertext $(c_0, \ldots, c_k)$ that can be decrypted as was shown in Section 4.1, relinearization is supposed to produce a ciphertext $(c'_0, \ldots, c'_{k-1})$ of size k, or – when applied repeatedly – of any size at least 2, that can be decrypted using a smaller degree decryption function to yield the same result. This conversion will require a so-called *evaluation key* (or *keys*) to be given to the evaluator, as we will explain below.

In FV, suppose we have a size 3 ciphertext (c_0, c_1, c_2) that we want to convert into a size 2 ciphertext (c'_0, c'_1) that decrypts to the same result. Suppose we are also given a pair a pair $\texttt{evk} = \big([-(as+e)+s^2]_q, a\big)$, where $a \xleftarrow{\$} R_q$, and $e \leftarrow \chi$. Now set $c'_0 = c_0 + \texttt{evk}[0]c_2$, $c'_1 = c_1 + \texttt{evk}[1]c_2$, and define the output to be the pair (c'_0, c'_1). Interpreting this as a size 2 ciphertext and decrypting it yields

$$c'_0 + c'_1 s = c_0 + (-(as+e)+s^2)c_2 + c_1 s + ac_2 s = c_0 + c_1 s + c_2 s^2 - ec_2\,.$$

This is almost what is needed, i.e. $c_0 + c_1 s + c_2 s^2$ (see Section 4.1), except for the additive extra term ec_2. Unfortunately, since c_2 has coefficients up to size q, this extra term will make the decryption process fail.

Instead we use the classical solution of writing c_2 in terms of some smaller base w (see e.g. [8, 7, 5, 15]) as $c_2 = \sum_{i=0}^{\ell} c_2^{(i)} w^i$. Instead of having just one evaluation key (pair) as above, suppose we have $\ell+1$ such pairs constructed as in Section 3.3. Then one can show that instead setting c'_0 and c'_1 as in Section 3.3 successfully replaces the large additive term that appeared in the naive approach above with a term of size linear in w.

This same idea can be generalized to relinearizing a ciphertext of any size $k+1$ to size $k \geq 2$, as long as a generalized set of evaluation keys is generated in the $\texttt{EvaluationKeyGen}(\texttt{sk}, w)$ function. Namely, suppose we have a set of evaluation keys $\texttt{evk}_2$ (corresponding to s^2), $\texttt{evk}_3$ (corresponding to s^3) and so on up to $\texttt{evk}_k$ (corresponding to s^k), each generated as in Section 3.3. Then relinearization converts $(c_0, c_1, \ldots, c_k)$ into $(c'_0, c'_1, \ldots, c'_{k-1})$, where

$$c'_0 = c_0 + \sum_{i=0}^{\ell} \texttt{evk}_k[i][0]c_k^{(i)}\,,$$

$$c'_1 = c_1 + \sum_{i=0}^{\ell} \texttt{evk}_k[i][1]c_k^{(i)}\,,$$

and $c'_j = c_j$ for $2 \leq j \leq k-1$.

Note that in order to generate evaluation keys, one needs to access the secret key, and so in particular the evaluating party would not be able to do this. The owner of the secret key must generate an appropriate number of evaluation keys and pass these to the evaluating party in advance of the relinearization computation. This means that the evaluating party should inform the key generating party beforehand whether or not they intend to relinearize, and if so, by how many steps. Note that if they choose to relinearize after every multiplication only one evaluation key, $\texttt{evk}_2$, is needed.

In SEAL v2.1 we define the function `Relinearize` (or rather family of functions) to mean this generalization of the second step of multiplication as was described in Section 3.3. It is implemented as `Evaluator::relinearize`. Suppose a ciphertext `ct` has size K and $L \in [2, K)$ is an integer, then `relinearize(ct,L)` returns a ciphertext of size L encrypting the same message as `ct`.

4.4 Addition

We also need to generalize addition to be able to operate on ciphertexts of any size. Suppose we have two SEAL v2.1 ciphertexts $\texttt{ct}_1 = (c_0, \ldots, c_j)$ and $\texttt{ct}_2 = (d_0, \ldots d_k)$, encrypting plaintext polynomials m_1 and m_2, respectively. Suppose WLOG $j \leq k$. Then

$$\texttt{ct}_{\text{add}} = ([c_0 + d_0]_q, \ldots, [c_j + d_j]_q, d_{j+1}, \ldots, d_k)$$

encrypts $[m_1 + m_2]_t$. Subtraction works exactly analogously.

In SEAL v2.1 we define the functions `Add` (or rather family of functions) to mean this generalization of addition. It is implemented as `Evaluator::add`. We also provide a function `Sub` for subtraction, which works in an analogous way, and is implemented as `Evaluator::sub`.

4.5 Other Homomorphic Operations

In SEAL v2.1 we provide a function `Negate` to perform homomorphic negation. This is implemented in the library as `Evaluator::negate`.

We also provide the functions $\texttt{AddPlain}(\texttt{ct}, m_{\text{add}})$ and $\texttt{MultiplyPlain}(\texttt{ct}, m_{\text{mult}})$ that, given a ciphertext `ct` encrypting a plaintext polynomial m, and unencrypted plaintext polynomials $m_{\text{add}}, m_{\text{mult}}$, output encryptions of $m + m_{\text{add}}$ and $m \cdot m_{\text{mult}}$, respectively. When one of the operands in either addition or multiplication does not need to be protected, these operations can be used to hugely improve performance over first encrypting the plaintext and then performing the normal homomorphic addition or multiplication. We will also see later in Section 5 that `MultiplyPlain` incurs much less noise to the ciphertext than normal `Multiply`, which will allow the evaluator to perform significantly more `MultiplyPlain` than `Multiply` operations. These functions are implemented in SEAL v2.1 as `Evaluator::add_plain` and `Evaluator::multiply_plain`. Analogously to `AddPlain` we have implemented a plaintext subtraction function as `Evaluator::sub_plain`.

In many situations it is necessary to multiply together several ciphertexts homomorphically. The naive sequential way of doing this has very poor noise growth properties. Instead, the user should use a low-depth arithmetic circuit. For homomorphic addition of several values the exact method for doing so is less important. SEAL v2.1 defines functions `MultiplyMany` and `AddMany`, which either multiply together or add together several ciphertexts in an optimal way. These are implemented as `Evaluator::multiply_many` and `Evaluator::add_many`.

SEAL v2.1 has a faster algorithm for computing the `Square` of a ciphertext. The difference is only in computational complexity, and the noise growth behavior is the same as in calling `Evaluator::multiply` with a repeated input parameter. `Square` is implemented as `Evaluator::square`.

Exponentiating a ciphertext to a non-zero power should be done using a similar low-depth arithmetic circuit that `MultiplyMany` uses. We denote this function by `Exponentiate`, and implement it as `Evaluator:exponentiate`. The implementations of both `MultiplyMany` and `Exponentiate` relinearize the ciphertext down to size 2 after every multiplication. It is the responsibility of the user to create enough evaluation keys beforehand to ensure that these operations can be done.

4.6 Key Distribution

In Section 4.3 we already explained how key generation in SEAL v2.1 differs from textbook-FV. There is another subtle difference, that is also worth pointing out. In textbook-FV the

secret key is a polynomial sampled uniformly from R_2, i.e. it is a polynomial with coefficients in $\{0, 1\}$. In SEAL v2.1 we instead sample the key uniformly from R_3, i.e. we use coefficients $\{-1, 0, 1\}$.

5 Inherent Noise

In this section we explain the concept of *inherent noise*, or *ciphertext noise*. We will explain how the noise grows in homomorphic operations, presenting practical estimates that are used by `SimulationEvaluator` and by the automatic parameter selection module to help the user determine appropriate optimized parameters for their computation (see Section 7.5). Although in textbook-FV all ciphertexts have size 2, in SEAL v2.1 we allow ciphertexts of any size greater than or equal to 2. We give general bounds accordingly.

Definition 1 (Inherent noise). *Let* $\mathtt{ct} = (c_0, c_1, \ldots, c_k)$ *be a ciphertext encrypting the message* $m \in R_t$*. Its inherent noise is the unique polynomial* $v \in R$ *with smallest infinity norm such that*

$$\mathtt{ct}(s) = c_0 + c_1 s + \cdots + c_k s^k = \Delta m + v + aq$$

for some polynomial a*.*

We will often refer to both v and its infinity norm $\|v\|$ as *inherent noise*. We will see below in Section 5.1 that in fact $\|v\|$ is what matters the most, and in particular in the comments in the code $\|v\|$ is what inherent noise always refers to. In SEAL v2.1 the quantity $\|v\|$ is output by the function `Decryptor::inherent_noise`. To instead return the significant bit count of $\|v\|$, use the function `Decryptor::inherent_noise_bits`.

5.1 Maximal Noise

The main result related to inherent noise is that once it reaches a large enough value the ciphertext becomes corrupted and impossible to decrypt even with the correct secret key. The upper bound on the inherent noise depends on both the coefficient modulus q and the plaintext modulus t.

Lemma 1. *The function (or family of functions)* `Decrypt`*, as presented in Section 4.1, correctly decrypts a ciphertext as long as the inherent noise satisfies* $\|v\| < \Delta/2$*.*

Proof. Consider a ciphertext $\mathtt{ct} = (c_0, c_1, \ldots, c_k)$. Its decryption m' under a secret key s is defined as

$$m' = \left[\left\lfloor \frac{t}{q} \left[c_0 + c_1 s + \cdots + c_k s^k \right]_q \right\rceil \right]_t .$$

By definition of inherent noise, $c_0 + c_1 s + \cdots + c_k s^k = \Delta m + v \pmod q$, so

$$m' = \left[\left\lfloor \frac{t(\Delta m + v)}{q} \right\rceil \right]_t = \left[\left\lfloor m - \frac{r_t(q)}{q} m + \frac{t}{q} v \right\rceil \right]_t ,$$

where we used $q = \Delta t + r_t(q)$. This means that $m' = m$ in R_t as long as the terms $-r_t(q)m/q + tv/q$ are removed by the rounding. In other words, we need

$$\left\| -\frac{r_t(q)}{q} m + \frac{t}{q} v \right\| < \frac{1}{2} .$$

Since

$$\left\| -\frac{r_t(q)}{q} m + \frac{t}{q} v \right\| \leq \frac{t}{q} \cdot \|m\| + \frac{t}{q} \|v\| \leq \frac{t^2}{q} + \frac{t}{q} \|v\|,$$

it suffices to require that

$$\frac{t^2}{q} + \frac{t}{q} \|v\| < \frac{1}{2},$$

which can be written as

$$\|v\| < \frac{q}{2t} - t = \frac{\Delta}{2} + \frac{r_t(q)}{2t} - t < \frac{\Delta}{2}.$$

□

The noise bound $\Delta/2$ is output by `EncryptionParameters::inherent_noise_max`. To instead return the significant bit count of $\Delta/2$, use the function `EncryptionParameters::inherent_noise_bits_max`.

5.2 Overview of Noise Growth

We now present an overview of how the user can expect to noise behave in homomorphic operations in typical applications.

Since the results presented here are probabilistic estimates, it might be possible to construct examples of plaintexts and ciphertexts that yield very different looking results. Nevertheless, the estimates are vastly simpler to read and interpret than exact formulas, and can be expected to be accurate in most cases.

The noise growth estimates are presented in Table 2. For each operation we describe the output noise in terms of the noises of the inputs and the encryption parameters (recall Table 1). For input ciphertexts $\texttt{ct}_i$ we always denote their respective inherent noises by v_i. When there is a single encrypted input `ct` we denote its inherent noise by v.

We would like to take this opportunity to point out a few important facts about noise growth that the user should keep in mind.

1. Every ciphertext, even if it is freshly encrypted, contains a non-zero amount of noise.
2. In FV the noise in a freshly encrypted ciphertext depends only on the degree n of the polynomial modulus and a bound B on the output of the error distribution χ.
3. Addition and subtraction have small impact on noise. Note that $r_t(q) < t$, and in practice we almost always have $t \ll \Delta/2$.
4. Plain multiplication increases the noise by a constant factor that depends on the plaintext multiplier m. If integer encoders are used, N (the number of nonzero coefficients of m) and especially $\|m\|$ can be small, in which case the increase in noise can be just a few bits. When `PolyCRTBuilder` is used the situation is radically different as $N \approx n$ and $\|m\| \approx t$, and plain multiplication results in roughly the same kind of noise growth as normal multiplication.
5. Roughly speaking, multiplication increases the noise by a multiplicative factor of t when integer encoders are used, and by a factor of nt when `PolyCRTBuilder` is used. However, there is an additional multiplicative factor that depends in an exponential way on the sizes of the ciphertexts. When relinearization is used, the sizes never grow too large and this factor becomes largely insignificant. However, if relinearization is *not* used, it can easily become the dominant factor in the noise of the result. In addition to performance increases from having smaller ciphertext sizes, this gives another good reason to perform relinearization.

Operation	Input description	Estimated output noise
`Encrypt`	Plaintext $m \in R_t$	$2B\sqrt{2n/3}$
`Negate`	Ciphertext `ct`	$\|v\|$
`Add`/`Sub`	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$	$\|v_1\| + \|v_2\| + r_t(q)$
`AddPlain`/`SubPlain`	Ciphertext `ct` and plaintext m	$\|v\| + r_t(q)$
`MultiplyPlain`	Ciphertext `ct` and plaintext m with N non-zero coefficients	$N\|m\|\,(\|v\| + r_t(q)/2)$
`Multiply` (with integer encoders)	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$ of sizes j_1+1 and j_2+1	$t\,(\|v_1\| + \|v_2\| + r_t(q)) \times \left\lceil \sqrt{2n/3} \right\rceil^{j_1+j_2-1} 2^{j_1+j_2}$
`Multiply` (with `PolyCRTBuilder`)	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$ of sizes j_1+1 and j_2+1	$nt\,(\|v_1\| + \|v_2\| + r_t(q)) \times \left\lceil \sqrt{2n/3} \right\rceil^{j_1+j_2-1} 2^{j_1+j_2}$
`Square`	Ciphertext `ct` of size j	Same as `Multiply`(`ct`, `ct`) but faster
`Relinearize`	Ciphertext `ct` of size K and target size L such that $2 \leq L < K$	$\|v\| + (K-L)\sqrt{n}B(\ell+1)w$
`AddMany`	Ciphertexts $\texttt{ct}_1, \ldots, \texttt{ct}_k$	$\sum_i \|v_i\| + (k-1)r_t(q)$
`MultiplyMany`	Ciphertexts $\texttt{ct}_1, \ldots, \texttt{ct}_k$	Apply `Multiply` in a tree-like manner, and `Relinearize` down to size 2 after every multiplication
`Exponentiate`	Ciphertext `ct` and exponent k	Apply `MultiplyMany` to k copies of `ct`

Table 2: Noise estimates for homomorphic operations in SEAL.

6. Relinearization increases the noise only by an additive factor $c\Delta_l$, where c is a constant determined by the encryption parameters, and Δ_l equals the difference in ciphertext lengths before and after Relinearization. This should be contrasted with multiplication, which increases the noise by a multiplicative factor. This means, for example, that after a few multiplications have been performed so that the noise has reached a size larger than the additive factor, relinearization no longer has a large impact on noise. Instead, it will only be beneficial due to the smaller noise increase in subsequent multiplications (see above). On the other hand, relinearizing after the very first multiplication is typically not an optimal strategy due to the additive factor being significantly larger than the noise resulting purely from multiplications. Subsequent multiplications will then build more noise on top of the (relatively large) additive factor that came from relinearization.
7. The decomposition bit count (recall Table 1) has a significant effect on both performance (recall Section 4.3) and noise growth in relinearization. Tuning down the decomposition bit count has a positive impact on noise growth in relinearization, and a negative impact on the computational cost of relinearization. However, when the entire computation is considered, it is not obvious at all what an optimal decomposition bit count should be, and at which points in the computation relinearization should be performed. Optimizing these choices is a difficult task and an interesting research problem. We have included several examples in the code to illustrate the situation, and we recommend the user to experiment to get a good understanding of how relinearization behaves.

6 Encoding

One of the most important aspects in making homomorphic encryption practical and useful is in using an appropriate *encoder* for the task at hand. Recall from Section 3 that plaintext elements in the FV scheme are polynomials in R_t (represented in SEAL as `BigPoly` objects), and homomorphic operations on ciphertexts are reflected in the plaintext side as corresponding (multiplication and addition) operations in the ring R_t. In typical applications of homomorphic encryption the user would instead want to perform computations on integers (or real numbers), and encoders are responsible for converting these integer (or real number) inputs to elements of R_t in an appropriate way.

It is easy to see that encoding is a highly non-trivial task. The rings $\mathbb{Z}$ and R_t are very different (most obviously the set of integers is infinite, whereas R_t is finite), and they are certainly not isomorphic. However, typically one does not need the power to encrypt *any* integer, so we can just as well settle for some finite reasonably large subset of $\mathbb{Z}$ and try to find appropriate maps from that subset into R_t. But again there is a problem, because no non-trivial subset of $\mathbb{Z}$ is closed under additions and multiplications, so we have to settle for something that does not respect an arbitrary number of homomorphic operations. It is then the responsibility of the evaluating party to be aware of the type of encoding that is used, and perform only operations such that the underlying plaintexts throughout the computation remain possible to decode.

6.1 Scalar Encoder

Perhaps the simplest possible encoder is what we could call the *scalar encoder*. Given an integer a, simply encode it as the constant polynomial $a \in R_t$. Obviously we can only encode integers modulo t in this manner. Decoding amounts to reading the constant coefficient of the polynomial and interpreting that as an integer. The problem is that as soon as the underlying

plaintext polynomial (constant) wraps around t at any point during the computation, we are no longer doing integer arithmetic, but rather modulo t arithmetic, and decoding might yield an unexpected result. This means that t must be chosen to be possibly very large, which creates problems with the noise growth for two reasons. First, recall that the noise ceiling is $\Delta/2$, where $\Delta = \lfloor q/t \rfloor$, which decreases when t increases. Second, recall from Table 2 that the noise growth in most of the operations, and particularly in multiplication, depends strongly on t, so increasing t even a little bit can possibly significantly reduce the amount of noise that is available for homomorphic computations.

One possible way around this is to encrypt the integer twice, using two or more relatively prime plaintext moduli $\{t_i\}$. Then if the computation is done separately to each of the encryptions, in the end after decryption the result can be combined using the Chinese Remainder Theorem to yield an answer modulo $\prod t_i$. As long as this product is larger than the largest underlying integer appearing during the computation, the result will be correct as an integer.

The scalar encoder is currently *not* implemented in SEAL v2.1. Instead, it can be constructed as a special case of some of the other encoders by choosing their parameters in a certain way. In most practical applications the scalar encoder is not a good choice, as it is extremely wasteful in the sense that the entire huge plaintext polynomial is used to encode and encrypt only one small integer. The other encoders attempt to make better use of the plaintext polynomials by either packing more data into one polynomial, or spreading the data around inside the polynomial to obtain encodings with smaller coefficients.

6.2 Integer Encoder

In SEAL v2.1 the *integer encoder* is used to encode integers in a much more efficient manner than what the scalar encoder (Section 6.1) could do. The integer encoder is really a *family* of encoders, one for each integer base $B \geq 2$. We start by explaining how the integer encoder works with $B = 2$, and then comment on the general case, which is an obvious extension.

When $B = 2$, the idea of the integer encoder is to encode an integer $-(2^n - 1) \leq a \leq 2^n - 1$ as follows. First, form the (up to n-bit) binary expansion of $|a|$, say $a_{n-1} \dots a_1 a_0$. Then the binary encoding of a is

$$\texttt{IntegerEncode}(a, B = 2) = \text{sign}(a) \cdot \left(a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \right) .$$

Remark 3. In SEAL v2.1 we only have an unsigned big integer data type (`BigUInt`), so we represent each coefficient of the polynomial as an unsigned integer modulo t. For example, the -1 coefficients of the polynomial will be stored as the unsigned integers $t - 1$.

Decoding (`IntegerDecode`) amounts to evaluating the plaintext polynomial at $x = 2$. It is clear that in good conditions (see below) the integer encoder respects integer operations:

$$\texttt{IntegerDecode}\left[\texttt{IntegerEncode}(a, B = 2) + \texttt{IntegerEncode}(b, B = 2)\right] = a + b\,,$$

$$\texttt{IntegerDecode}\left[\texttt{IntegerEncode}(a) \cdot \texttt{IntegerEncode}(b, B = 2)\right] = ab\,.$$

When the integer encoder with $B = 2$ is used, the norms of the plaintext polynomials are guaranteed to be bounded by 1 only when no homomorphic operations have been performed. When two such encodings are added together, the coefficients sum up and can therefore get bigger. In multiplication this is even more noticeable due to the appearance of cross terms. In multiplications the polynomial length also grows, but often in practice this is not an issue

due to the large number of coefficients available in the plaintext polynomials. Things will go wrong as soon as *any* modular reduction – either modulo the polynomial modulus $x^n + 1$, or modulo the plaintext modulus t – occurs in the underlying plaintexts at any point during the computation. If this happens, decoding will yield an incorrect result, but there will be no other indication that something has gone wrong. It is therefore crucial that the evaluating party understands the limitations of the integer encoder, and makes sure that the plaintext underlying the result ciphertext will still be possible to decode correctly.

When B is set to some integer larger than 2, instead of a binary expansion (as was done in the example above) a base-B expansion is used, where the coefficients are chosen from the symmetric set $[-(B-1)/2, \ldots, (B-1)/2]$. There is a unique such representation with at most n coefficients for each integer in $[-(B^n-1)/2, (B^n-1)/2]$. Decoding is obviously performed by evaluating a plaintext polynomial at $x = B$. Note that with $B = 3$ the integer encoder provides encodings with equally small norm as with $B = 2$, but with a more compact representation, as it does not waste space in repeating the sign for each non-zero coefficient. Larger B provide even more compact representations, but at the cost of increased coefficients. In most common applications taking $B = 2$ or 3 is a good choice, and there is little difference between these two.

The integer encoder is significantly better than the scalar encoder, as the coefficients in the beginning are much smaller than in plaintexts encoded with the scalar encoder, leaving more room for homomorphic operations before problems with reduction modulo t are encountered. From a slightly different point of view, the binary encoder allows a smaller t to be used, resulting in both smaller noise growth in homomorphic operations, and a larger noise ceiling.

The integer encoder is available in SEAL v2.1 through the class `IntegerEncoder`. Its constructor will require both the `plain_modulus` and the base B as parameters. If no base is given, the default value $B = 2$ is used.

Binary and balanced encoders In earlier versions of the library the integer encoder was instead exposed through two classes, the `BinaryEncoder` and the `BalancedEncoder`. These classes still exist, and are used under the hood by `IntegerEncoder`. In future releases we might prevent the user from creating them directly, so at this point it is recommended to start using `IntegerEncoder` instead.

6.3 Fractional Encoder

There are several ways for encoding rational numbers. The simplest and often most efficient way is to simply scale all rational numbers to integers, encode them using the integer encoder described above, and modify any computations to instead work with such scaled integers. After decryption and decoding the result needs to be scaled down by an appropriate amount. While efficient, in some cases this technique can be annoying, as it will require one to always keep track of how each plaintext has been scaled. Here we describe what we call the *fractional encoder*. Just like the integer encoder (Section 6.2 above), the fractional encoder is a family of encoders, parametrized by an integer base $B \geq 2$. The function of this base is exactly the same as in the integer encoder, so since the generalization is obvious, we will only explain how the fractional encoder works when $B = 2$.

The easiest way to explain how the fractional encoder (with $B = 2$) works is through a simple example. Consider the rational number 5.8125. It has a finite binary expansion

$$5.875 = 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4}\,.$$

First we take the integer part and encode it as usual with the integer encoder, obtaining the polynomial $\texttt{IntegerEncode}(5, B = 2) = x^2 + 1$. Then we take the fractional part $2^{-1} + 2^{-2} + 2^{-4}$, add n (recall Table 1) to each exponent, and convert it into a polynomial by changing the base 2 into the variable x, resulting in $x^{n-1} + x^{n-2} + x^{n-4}$. Next we flip the signs of each of the terms, in this case obtaining $-x^{n-1} - x^{n-2} - x^{n-4}$. For rational numbers r in the interval $[0, 1)$ with finite binary expansion we denote this encoding by $\texttt{FracEncode}(r, B = 2)$. For any rational number r with finite binary expansion we set

$$\texttt{FracEncode}(r, B = 2) = \text{sign}(r) \cdot [\texttt{IntegerEncode}(\lfloor |r| \rfloor, B = 2) + \texttt{FracEncode}(\{|r|\}, B = 2)] \,,$$

where $\{\cdot\}$ denotes the fractional part. For example,

$$\texttt{FracEncode}(5.8125, B = 2) = -x^{n-1} - x^{n-2} - x^{n-4} + x^2 + 1 \,.$$

Decoding works by essentially reversing the steps described above. First, separate the high-degree part of the plaintext polynomial that describes the fractional part. Next invert the signs of those terms and shift their exponents by $-n$. Finally evaluate the entire expression at $x = 2$. We denote this operation $\texttt{FracDecode}(\cdot, B = 2)$.

It is not hard to see why this works. As a very simple example, imagine computing $1/2 \cdot 2$, where $\texttt{FracEncode}(1/2, B = 2) = -x^{n-1}$ and $\texttt{FracEncode}(2, B = 2) = x$. Then in the ring R_t we have

$$\texttt{FracEncode}(1/2, B = 2) \cdot \texttt{FracEncode}(2, B = 2) = -x^n = 1 \,,$$

which is exactly what we would expect, as $\texttt{FracDecode}(1, B = 2) = 1$. For a more complicated example, consider computing $5.8125 \cdot 2.25$. We already computed $\texttt{FracEncode}(5.8125, B = 2)$ above, and $\texttt{FracEncode}(2.25, B = 2) = -x^{n-2} + x$. Then

$$\begin{aligned}
&\texttt{FracEncode}(5.8125, B = 2) \cdot \texttt{FracEncode}(2.25, B = 2) \\
&= (-x^{n-1} - x^{n-2} - x^{n-4} + x^2 + 1) \cdot (-x^{n-2} + x) \\
&= x^{2n-3} + x^{2n-4} + x^{2n-6} - 2x^n - x^{n-1} - x^{n-2} - x^{n-3} + x^3 + x \\
&= -x^{n-1} - x^{n-2} - 2x^{n-3} - x^{n-4} - x^{n-6} + x^3 + x + 2 \,.
\end{aligned}$$

Finally,

$$\begin{aligned}
&\texttt{FracDecode}(-x^{n-1} - x^{n-2} - 2x^{n-3} - x^{n-4} - x^{n-6} + x^3 + x + 2, B = 2) \\
&= \left[x^3 + x + 2 + x^{-1} + x^{-2} + 2x^{-3} + x^{-4} + x^{-6}\right]_{x=2} = 13.078125 \,.
\end{aligned}$$

There are several important aspects of the fractional encoder that require further clarification. First of all, above we described only how $\texttt{FracEncode}(\cdot, B = 2)$ works for rational numbers that have finite binary expansion, but many rational numbers do not, in which case we need to truncate the expansion of the fractional part to some precision, say n_f bits (equivalently, high-degree coefficients of the plaintext polynomial). Next, the decoding process needs to somehow know which coefficients of the plaintext polynomial should be interpreted as belonging to the fractional part and which to the integer part. For this purpose we fix a number n_i to denote the number of coefficients reserved for the integer part, and all of the remaining $n - n_i$ coefficients will be interpreted as belonging to the fractional part. Note that $n_f + n_i \leq n$, and that n_f only matters in the encoding process, whereas n_i is needed both in encoding (can only encode integer parts up to n_i bits) and in decoding.

Decoding can fail for two reasons. First, if any of the coefficients of the underlying plaintext polynomials wrap around the plaintext modulus t the result after decoding is likely to be incorrect, just as in the normal integer encoder (recall Section 6.2). Second, homomorphic multiplication will cause the fractional parts of the underlying plaintext polynomials to expand down towards the integer part, and the integer part to expand up towards the fractional part. If these different parts get mixed up, decoding will fail. Typically the user will want to choose n_f to be as small as possible, as many rational numbers will have dense infinite expansions, filling up most of the leading n_f coefficients. When such polynomials are multiplied, cross terms cause the coefficients to quickly increase in size, resulting in them getting reduced modulo t unless t is chosen to be very large.

When B is set to some integer larger than 2, instead of a binary expansion (as was done in the example above) a base-B expansion is used, where the coefficients are chosen from the symmetric set $[-(B-1)/2, \ldots, (B-1)/2]$. Again, in this case decoding amounts to evaluating polynomials $x = B$.

The fractional encoder is available in SEAL v2.1 through the classe `FractionalEncoder`. Its constructor will require the `plain_modulus`, the base B, and positive integers n_f and n_i with $n_f + n_i \leq n$ as parameters. If no base is given, the default value $B = 2$ is used.

Binary and balanced fractional encoders In earlier versions of the library the fractional encoder was instead exposed through two classes, the `BinaryFractionalEncoder` and the `BalancedFractionalEncoder`. These classes still exist, and are used under the hood by `FractionalEncoder`. In future releases we might prevent the user from creating them directly, so at this point it is recommended to start using `FractionalEncoder` instead.

6.4 CRT Batching

The last encoder that we describe is very different from the previous ones, and extremely powerful. It allows the user to pack n integers modulo t into one plaintext polynomial, and to operate on those integers in a *SIMD (Single Instruction, Multiple Data)* manner. This technique is often called *batching* in homomorphic encryption literature. For more details and applications we refer the reader to [6, 25].

Batching only works when the plaintext modulus t is chosen to be a prime number and congruent to 1 (mod $2n$), which we assume to be the case[6]. In this case the multiplicative group of integers modulo t contains a subgroup of size $2n$, which means that there is an integer $\zeta \in \mathbb{Z}_t$ such that $\zeta^{2n} = 1 \pmod{t}$, and $\zeta^m \neq 1 \pmod{t}$ for all $0 < m < 2n$. Such an element ζ is called a *primitive $2n$-th root of unity modulo t*. Having a primitive $2n$-th root of unity in $\mathbb{Z}_t$ is important because then the polynomial modulus $x^n + 1$ factors modulo t as

$$x^n + 1 = (x - \zeta)(x - \zeta^3) \ldots (x - \zeta^{2n-1}) \pmod{t},$$

and according to the *Chinese Remainder Theorem (CRT)* the ring R_t factors as

$$R_t = \frac{\mathbb{Z}_t[x]}{(x^n+1)} = \frac{\mathbb{Z}_t[x]}{\prod_{i=0}^{n-1}(x - \zeta^{2i+1})} \overset{\text{CRT}}{\cong} \prod_{i=0}^{n-1} \frac{\mathbb{Z}_t[x]}{(x - \zeta^{2i+1})} \cong \prod_{i=0}^{n-1} \mathbb{Z}_t[\zeta^{2i+1}] \cong \prod_{i=0}^{n-1} \mathbb{Z}_t .$$

All of the isomorphisms above are isomorphisms of rings, which means that they respect both the multiplicative and additive structures on both sides, and allows one to perform n

[6] Note that this means $t > 2n$, which can in some cases turn out to be an annoying limitation.

coefficient-wise additions (resp. multiplications) in integers modulo t (right-hand side) at the cost of one single addition (resp. multiplication) in R_t (left-hand side). It is easy to describe explicitly what the isomorphisms are. For simplicity, denote $\alpha_i = \zeta^{2i+1}$. In one direction the isomorphism is given by

$$\texttt{Decompose} : R_t \xrightarrow{\cong} \prod_{i=0}^{n-1} \mathbb{Z}_t \,,\; m(x) \longmapsto [m(\alpha_0), m(\alpha_1), \ldots, m(\alpha_{n-1})] \,.$$

The inverse is slightly tricker to describe, so we omit it here for the sake of simplicity. We define `Compose` to be the inverse of `Decompose`. In SEAL v2.1, these isomorphisms are computed using a negacylic variant of the Number Theoretic Transform (NTT).

When used correctly, batching can provide an enormous performance improvement over the other encoders. When using batching for computations on encrypted integers rather than on integers modulo t, one needs to ensure that the values in the *slots* never wrap around t during the computation. Note that this is exactly the same limitation the scalar encoder has (recall Section 6.1), and could be solved by choosing t to be large enough, which will unfortunately cause large noise growth and reduce the noise ceiling.

SEAL v2.1 provides all of the batching-related tools in the `PolyCRTBuilder` class. The constructor of `PolyCRTBuilder` takes an instance of `EncryptionParameters` as argument, and will throw an exception unless the parameters are appropriate, as was described in the beginning of this section.

7 Encryption Parameters

Everything in SEAL v2.1 starts with the construction of an instance of a container that holds the encryption parameters (`EncryptionParameters`). This will store the parameters $x^n + 1$ (`poly_modulus`), q (`coeff_modulus`), t (`plain_modulus`), σ (`noise_standard_deviation`), B (`noise_max_deviation`), $\log w$ (`decomposition_bit_count`), and a source of randomness (`random_generator`). Some of these parameters are optional, e.g. if the user does not specify σ or B they will be set to default values. If the user does not set the decomposition bit count, SEAL will assume that no relinearization is going to be performed, and prevents the creation of any evaluation keys (recall Section 1.2 and Section 4.3). If no randomness source is given, SEAL will automatically use `std::random_device`.

Remark 4. The choice of encryption parameters significantly affects the performance, capabilities, and security of the encryption scheme. Some choices of parameters may be insecure, give poor performance, yield ciphertexts that will not work with any homomorphic operations, or a combination of all of these.

In this section we will describe the encryption parameters and their impact on performance. We will discuss security briefly in Section 8. In Section 7.5 we will discuss the automatic parameter selection tools in SEAL v2.1, which can assist the user in determining (close to) optimal encryption parameters for certain use-cases.

7.1 Default Values

Unlike in previous versions of SEAL, the constructor of `EncryptionParameters` sets the values for σ and B by default to the ones returned by the static functions

`ChooserEvaluator::default_noise_standard_deviation()`

and

`ChooserEvaluator::default_noise_max_deviation()`.

Currently these default values are set to 3.19 and 15.95, respectively, but it should be easy for a user to change them if they desire to.

As we have mentioned several times before, the user no longer needs to set a value for `decomposition_bit_count` unless they choose to use relinearization. By default the constructor will set this value to zero, which will prevent the construction of evaluation keys.

SEAL v2.1 contains a list of pairs (n, q) that are returned by the static function

`ChooserEvaluator::default_parameter_options()`.

The list that is currently used by default is presented in Table 3.

n	1024	2048	4096	8192	16384
q	$2^{35} - 2^{14} + 2^{11} + 1$	$2^{60} - 2^{14} + 1$	$2^{116} - 2^{18} + 1$	$2^{226} - 2^{26} + 1$	$2^{435} - 2^{33} + 1$

Table 3: Default pairs (n, q).

7.2 Polynomial Modulus

The polynomial modulus (`poly_modulus`) should be a polynomial of the form $x^n + 1$, where n is a power of 2. This is both for security and performance reasons (see Section 8).

Using a larger n allows for a larger q to be used without decreasing the security level, which in turn increases the noise ceiling and thus allows for larger t to be used, which is often important for integer encodings to work (recall Section 6). Increasing n will significantly decrease performance, but on the other hand it will allow for more elements of $\mathbb{Z}_t$ to be batched into one plaintext when using `PolyCRTBuilder`.

7.3 Coefficient Modulus and Plaintext Modulus

Suppose the polynomial modulus is held fixed. Then the choice of the coefficient modulus q affects two things: the upper bound on the inherent noise that a ciphertext can contain[7] (see Section 5.1), and the security level[8] (see Section 8.2 and references therein).

In principle we can take q to be any integer, as long as it is not too large to cause security problems (see above). However, taking q to be of special form provides huge performance benefits, as we will now explain. First, if q is of the form $2^A - B$, where B is an integer of small absolute value, then modular reduction modulo q can be sped up, yielding overall better performance.

Next, if $2n|(q-1)$, SEAL can use the Number Theoretic Transform (NTT) for polynomial multiplications, resulting in huge performance benefits perhaps most importantly in relinearization and encryption. We use David Harvey's algorithm for NTT as described in [18], which additionally requires that $4q \leq \beta$, where β denotes the *word size*,

$$\beta = 2^{64\lceil \log(q)/64 \rceil}.$$

[7] Bigger q means higher noise bound (good).
[8] Bigger q means lower security (bad).

If both requirements are not met, SEAL v2.1 automatically uses slower algorithms.

Third, if $t|(q-1)$ (i.e. $r_t(q) = 1$), then the noise growth properties are improved in certain homomorphic operations (recall Table 2). In principle, the plaintext modulus t can be any integer, but choosing t to be a power of 2 makes it very easy to have this last property satisfied.

The default parameters of Table 3 satisfy all of these guidelines. They are prime numbers of the form $2^A - B$ where B is much smaller than 2^A. They are congruent to 1 modulo $2n$, and not too close to the word size boundary. Finally, $r_t(q) = 1$ for t that are reasonably large powers of 2, for example the default parameters for $n = 4096$ provide good performance for t a power of 2 up to 2^{18}.

Note that when using batching (recall Section 6.4) it will not be possible to have t be a power of 2, as t needs to instead be a prime of particular form. In this case the user can try to choose the entire triple (n, q, t) simultaneously so that $t = 1 \pmod{2n}$ and q satisfies as many of the good properties listed above as possible.

7.4 Encryption Parameter Qualifiers

Instances of the `EncryptionParameters` class are given as input to the constructors of tools such as `Encryptor` and `Decryptor`. These constructors then inspect the parameters, decide whether they are valid for use in SEAL, and which optimized algorithms they support. To make this process cleaner and visible to the user, SEAL v2.1 contains a struct called `EncryptionParameterQualifiers`, which is a dumb container for a small set of Boolean flags that describe critical features of the parameters. Given an instance of `EncryptionParameters`, the user can call `EncryptionParameters::get_qualifiers` to return an instance of the qualifiers struct, and inspect these flags. However, the only way to change the qualifiers is to change the encryption parameters themselves to support the particular features. Currently `EncryptionParameterQualifiers` contains 6 qualifiers, which are described in Table 4.

Qualifier	Description
`parameters_set`	`true` if the encryption parameters are valid for SEAL v2.1, otherwise `false`
`enable_relinearization`	`true` if `decomposition_bit_count` is positive, otherwise `false`
`enable_nussbaumer`	Describes whether Nussbaumer convolution [10] can be used for polynomial multiplication. This is `true` if `poly_modulus` is of the form $x^n + 1$, where n is a power of 2, and otherwise `false`. Note that in SEAL v2.1 this is necessarily `true` if `parameters_set` is `true`, as we only allow polynomial moduli of this form.
`enable_ntt`	`true` if NTT can be used for polynomial multiplication [18, 21], otherwise `false`. See Section 7.3 above for details.
`enable_batching`	`true` if batching (`PolyCRTBuilder`) can be used, otherwise `false`. See Section 6.4 for details.
`enable_ntt_in_multiply`	Not currently used.

Table 4: Encryption Parameter Qualifiers.

7.5 Automatic Parameter Selection

To assist the user in choosing parameters for a specific computation SEAL v2.1 provides an automatic parameter selection module. It consists of two parts: a `Simulator` component that simulates noise growth in homomorphic operations using the estimates of Table 2, and a `Chooser` component, which estimates the growth of the coefficients in the underlying plaintext

polynomials, and uses `Simulator` to simulate noise growth. `Chooser` also provides tools for computing an optimized parameter set once it knows what kind of computation the user wishes to perform.

Simulator `Simulator` consists of two components. A `Simulation` is a model of the inherent noise $\|v\|$ (recall Section 5) in a ciphertext. `SimulationEvaluator` is a tool that performs all of the usual homomorphic operations on simulations rather than on ciphertexts, producing new simulations with noise value computed according to Table 2. `Simulator` is implemented in SEAL v2.1 by the `Simulation` and `SimulationEvaluator` classes.

Chooser `Chooser` consists of three components. A `ChooserPoly` models a plaintext polynomial, which can be thought of as being either encrypted or unencrypted. In particular, it keeps track of two quantities: the largest coefficient in the plaintext (coefficient bound), and the number of non-zero coefficients in the plaintext (length bound). It also stores the *operation history* of the plaintext, which can involve encryption, and any number of homomorphic operations with an arbitrary number of other `ChooserPoly` objects as inputs. `ChooserPoly` also provides a tool for estimating the noise that would result when the operations stored in its operation history are performed, which it does using `Simulator`, and a tool for testing whether a given set of encryption parameters can support the computations in its history. `ChooserEvaluator` is a tool that performs all of the usual homomorphic operations on `ChooserPoly` objects rather than on ciphertexts, producing new `ChooserPoly` objects with coefficient bound and length bound estimates based on the operation in question, and on the inputs. Furthermore, `ChooserEvaluator` contains a tool for finding an optimized parameter set, which we will discuss below. `ChooserEncoder` creates a `ChooserPoly` that models an unencrypted plaintext (empty operation history), encoded using the integer encoder (recall Section 6.2). `ChooserEncryptor` converts `ChooserPoly` objects with empty operation history (modeling unencrypted plaintexts) into ones with operation history consisting only of encryption. These tools are all implemented in SEAL v2.1 by the `ChooserPoly`, `ChooserEvaluator`, `ChooserEncoder`, and `ChooserEncryptor` classes.

Parameter Selection One of the most important tools in `Chooser` is the `SelectParameters` functionality. It takes as input a `ChooserPoly`, a set `ParameterOptions` of pairs (n, q), a value for σ, and a value for B, and attempts to find an optimal pair $(n_{\text{opt}}, q_{\text{opt}})$ from `ParameterOptions`, together with an optimal value t_{opt}, and returns the triple $(n_{\text{opt}}, q_{\text{opt}}, t_{\text{opt}})$, along with an optimal value for the decomposition bit count if relinearization was used. It also sets the parameters σ and B (see below). `SelectParameters` is implemented in SEAL v2.1 by the function `ChooserEvaluator::select_parameters`.

Recall from Section 7.1 that SEAL v2.1 has an easy-to-access (and easy-to-modify) default list of pairs (n, q), and values for σ and B. The basic version of the function `ChooserEvaluator::select_parameters` uses these, but there is also an overload that lets the user pass their own values to be used instead. There is also a third kind of overload that takes several `ChooserPoly` objects as input and ensures that the parameters returned are large enough to support the operation histories of each of them.

The way the `ChooserEvaluator::select_parameters` function works is as follows. First it looks at the `ChooserPoly` input(s) it is given, and selects a t just large enough to be sure that all the computations can be done without reduction modulo t taking place in the plaintext

polynomials[9]. Next, it loops through each (n, q) pair available in the order they were given, and runs the `ChooserPoly::test_parameters` function every time until a set of parameters is found that gives enough room for the noise.

If the computation involved relinearization, things are a little bit trickier. Whenever a new pair (n, q) is selected, the decomposition bit count is set to be the smallest possible so that $\lfloor \log_w q \rfloor + 1 = 2$ (recall Table 1). This means that in relinearization the polynomial coefficients can be split into two base-w components, which offers the best performance at the cost of higher noise growth, as noise grows in relinearization by an additive factor proportional to w (recall Table 2). If these parameters fail, the decomposition bit count will be decremented until decryption is expected to succeed, or the decomposition bit count becomes so small that $\lfloor \log_w q \rfloor + 1 > 5$, in which case the outermost loop moves on to the next (n, q) pair. If eventually a good parameter set is found, the function populates the instance of `EncryptionParameters` given to it, and returns `true`. Otherwise it returns `false`. The SEALExamples project that comes with the code contains a detailed demonstration of using the parameter selection tools.

8 Security of FV

8.1 RLWE

The security of the FV encryption scheme is based on the apparent hardness of the famous *Ring Learning with Errors (RLWE)* problem [22]. We give a definition of the *decision*-RLWE problem appropriate to the rings that we use.

Definition 2 (Decision-RLWE). *Let n be a power of 2. Let $R = \mathbb{Z}[x]/(x^n + 1)$, and $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ for some integer q. Let s be a random element in R_q, and let χ be the distribution on R_q obtained by choosing each coefficient of the polynomial from a discrete Gaussian distribution over $\mathbb{Z}$. Denote by $A_{s,\chi}$ denote the distribution obtained by choosing $a \leftarrow R_q$ uniformly at random, choosing $e \leftarrow \chi$, and outputting $(a, [a \cdot s + e]_q)$. Decision-RLWE is the problem of distinguishing between the distribution $A_{s,\chi}$ and the uniform distribution on R_q^2.*

It is possible to prove that for certain parameters the decision-RLWE problem is as hard as solving certain famous lattice problems in the worst case. However, in practice the parameters that are used are not necessarily in the range where the reduction holds, and the reduction might be very difficult to perform in any case.

Remark 5. While it is possible to prove security results for certain choices of the polynomial modulus other than $x^n + 1$ for n a power of 2 (see [22, 13]), these proofs require the error terms e to be sampled from the distribution χ in a way very different from how SEAL does so. This is one reason why we only allow polynomial moduli of the form $x^n + 1$ for n a power of 2.

In practice an attacker will not have unlimited access to the oracle generating samples in the decision-RLWE problem, but the number of samples available will be limited to d. We call this the *d-sample decision-RLWE problem*. It is possible to prove that solving the d-sample decision-RLWE problem is equally hard as solving the $(d-1)$-sample decision-RLWE problem with the secret s instead sampled from the error distribution χ [23]. Furthermore, it is possible

[9] This makes sense in the context of the binary and balanced encoders. Currently automatic parameter selection is only designed to work with these integer encoders.

to argue [17, 15] that the security level remains roughly the same even if s is sampled from almost any narrow distribution with enough entropy, such as the uniform distribution on R_2 or R_3, as in SEAL v2.1 (recall Section 4.6).

It is easy to give an informal argument for the security of the FV scheme, assuming the hardness of decision-RLWE. Namely, the FV public key is indistinguishable from uniform based on the hardness of 2-sample decision-RLWE (or rather the hardness of the 1-sample small secret variant described above). Subsequently, an FV encryption is indistinguishable from uniform based on the 3-sample decision-RLWE (or rather the hardness of the 2-sample small secret variant described above), and the assumed uniformity of the public key. We refer the reader to [23] and [15] for further details and discussion.

8.2 Choosing Parameters for Security

Each RLWE sample $(as + e, a) \in R_q^2$ can be used to extract n *Learning with Errors (LWE)* samples [24, 20]. To the best of our knowledge, the most powerful attacks against d-sample RLWE all work by instead attacking the nd-sample LWE problem, and when estimating the security of a particular set of RLWE parameters it makes sense to instead focus on estimating the security of the induced set of LWE parameters.

At the time of writing this, determining the concrete hardness of parametrizations of (R)LWE is an active area of research (see e.g. [11, 9, 1]) and no standardized (R)LWE parameter sets exist. We strongly suggest that the user consults experts in the security of (R)LWE when choosing parameters for SEAL.

8.3 Circular Security

Recall from Section 3 that in textbook-FV we require an evaluation key, which is essentially a masking of the secret key raised to the power 2 (or, more generally, to some higher power). Unfortunately, it is not possible to argue the uniformity of the evaluation key based on the decision-RLWE assumption. Instead, one can think of it as an encryption of (some power of) the secret key *under the secret key itself*, and to argue security one needs to make the extra assumption that the encryption scheme is secure even when the adversary has access to all of the evaluation keys which may exist. In [15] this is assumption is noted as a form of *weak circular security*.

In SEAL v2.1 we do not perform relinearization by default, and therefore do not require the generation of evaluation keys, so it is possible to avoid having to use this extra assumption. However, in many cases using relinearization has massive performance benefits, and – as far as we are aware – there exist no known practical attacks that would exploit the evaluation keys.

8.4 Circuit Privacy

The privacy goal of SEAL is to allow the evaluation of arithmetic circuits on encrypted inputs, without revealing anything about the values of the input wires to the circuits beyond what is revealed by the output wires. In particular, no attempt is made to keep the arithmetic circuit itself private.

There are ways in which a semi-honest party can find information about a circuit that was evaluated on encrypted data simply by looking at the resulting ciphertexts, or – even better – at resulting ciphertext/plaintext pairs. For example, if no relinearization is used, they can

read the highest power that was computed from the size of the output ciphertext. Whoever holds the secret key can compute the noise in the ciphertext and deduce information about the structure of the circuit from that, especially if no relinearization was used.

It is possible to obtain circuit privacy in a couple of ways. One way already described by Gentry in [16] is to flood the noise by first relinearizing the ciphertext size down to 2, and then adding an encryption of 0 with noise super-polynomially larger than the old noise. This will statistically hide the old noise, but seriously restricts the number of homomorphic operations that can be performed. An alternative to this approach, replacing flooding with a *soak-spin-repeat* strategy, is given by Ducas and Stehlé in [14]. This technique restricts the scheme less, but uses Gentry's bootstrapping process to repeatedly re-encrypt the ciphertext. This is unfortunately slow, requires also a significant amount of room for noise, and is not currently implemented in SEAL. Finally, there are scheme specific circuit privacy techniques that can in some cases be much more efficient than the two generic method mentioned above. One such method for the GSW cryptosystem is described in [4].

References

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.

[2] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. Cryptology ePrint Archive, Report 2015/1192, 2015. `http://eprint.iacr.org/2015/1192`.

[3] 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.

[4] Florian Bourse, Rafaël Del Pino, Michele Minelli, and Hoeteck Wee. Fhe circuit privacy almost for free.

[5] 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.

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

[7] 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.

[8] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[9] Johannes A. Buchmann, Niklas Büscher, Florian Göpfert, Stefan Katzenbeisser, Juliane Krämer, Daniele Micciancio, Sander Siim, Christine van Vredendaal, and Michael Walter. Creating cryptographic challenges using multi-party computation: The LWE challenge. In Keita Emura, Goichiro Hanaoka, and Rui Zhang, editors, *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography, AsiaPKC@AsiaCCS, Xi'an, China, May 30 - June 03, 2016*, pages 11–20. ACM, 2016.

[10] Richard Crandall and Carl Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006.

[11] Eric Crockett and Chris Peikert. Challenges for ring-lwe. Cryptology ePrint Archive, Report 2016/782, 2016. `http://eprint.iacr.org/2016/782`.

[12] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. Technical report, Microsoft Research, 2015. `http://research.microsoft.com/apps/pubs/default.aspx?id=258435`.

[13] Léo Ducas and Alain Durmus. Ring-lwe in polynomial rings. In Marc Fischlin, Johannes A. Buchmann, and Mark Manulis, editors, *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 34–51. Springer, 2012.

[14] Léo Ducas and Damien Stehlé. Sanitization of FHE ciphertexts. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 294–310. Springer, 2016.

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

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

[17] Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Robustness of the learning with errors assumption. 2010.

[18] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014.

[19] Rachel Player Kim Laine. Simple encrypted arithmetic library - seal (v2.0). Technical report, September 2016.

[20] Tancrède 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] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. Cryptology ePrint Archive, Report 2016/504, 2016. http://eprint.iacr.org/2016/504.

[22] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[23] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 35–54. Springer, 2013.

[24] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

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