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

Abstract. Achieving fully homomorphic encryption was a longstanding open problem in cryptography until it was resolved by Gentry in 2009. Soon after, several homomorphic encryption schemes were proposed. The early homomorphic encryption schemes were extremely impractical, but recently new implementations, new data encoding techniques, and a better understanding of the applications have started to change the situation. In this paper we introduce the most recent version (v2.1) of Simple Encrypted Arithmetic Library - SEAL, a homomorphic encryption library developed by Microsoft Research, and describe some of its core functionality.

1 Introduction

In many traditional encryption schemes (e.g. RSA, ElGamal, Paillier) the plaintext and ciphertext spaces have a tremendous amount of algebraic structure, but the encryption and decryption functions either do not respect the algebraic structure at all, or respect only a part of it. Many schemes, such as ElGamal (resp. e.g. Paillier), are multiplicatively homomorphic (resp. additively homomorphic), but this restriction to one single algebraic operation is a very strong one, and the most interesting applications would instead require a ring structure between the plaintext and ciphertext spaces to be preserved by encryption and decryption. The first such encryption scheme was presented by Craig Gentry in his famous work [22], and since then researchers have introduced a number of new and more efficient *fully* homomorphic encryption schemes.

The early homomorphic encryption schemes were extremely impractical, but recently new implementations, new data encoding techniques, and a better understanding of the applications have started to change the situation. In 2015 we released the *Simple Encrypted Arithmetic Library - SEAL* [19] with the goal of providing a well-engineered and documented homomorphic encryption library, with no external dependencies, that

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

would be equally easy to use both by experts and by non-experts with little or no cryptographic background.
SEAL is written in C++11, and contains a .NET wrapper library for the public API. It comes with example projects demonstrating key features, written both in C++ and in C#. SEAL compiles and is tested on modern versions of Visual Studio and GCC. In this paper we introduce the most recent version, SEAL v2.1, and describe some of its core functionality. The library is publicly available at

`http://sealcrypto.codeplex.com`

and is licensed under the Microsoft Research License Agreement.

1.1 Related Work

A number of other libraries implementing homomorphic encryption exist, e.g. HElib [2] and $\Lambda \circ \lambda$ [18]. The FV scheme has been implemented in [9, 1], both of which use the ideal lattice library NFLlib [31]. Perhaps the most comparable work to SEAL is the C++ library HElib [2] which implements the BGV homomorphic encryption scheme [12].
A comparison of popular homomorphic encryption schemes, including BGV and FV, was presented by Costache and Smart in [14]. An comparison of the implementations, respectively, of BGV as in HElib and of FV as in SEAL would be very interesting, but appears challenging. One reason for this is that the documentation available for HElib [24, 25, 26] does not in general make clear how to select optimal parameters for performance, and in [26, Appendix A] it is noted '*[t]he BGV implementation in HElib relies on a myriad of parameters ... it takes some experimentation to set them all so as to get a working implementation with good performance*'. On the other hand, we know better how to select good parameters for performance for SEAL (see Section 4 below). Such a comparison is therefore deferred to future work.

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.

3 Implementing the Fan-Vercauteren Scheme

In this section we present our implementation of the Fan-Vercauteren (FV) scheme [20].

Parameter	Description	Name in SEAL
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 $\mathbb{Z}_a[x]/(x^n + 1)$	
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.

As described in [20], the FV scheme consists of the following algorithms: `SecretKeyGen`, `PublicKeyGen`, `EvaluateKeyGen`, `Encrypt`, `Decrypt`, `Add`, `Mul`, and `Relin` (version 1). In SEAL we generalize the scheme a little bit, as will be discussed below.

3.1 Plaintext Space and Encodings

In FV the plaintext space is the polynomial quotient ring $R_t = \mathbb{Z}_t[x]/(x^n + 1)$. 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. Plaintext polynomials are represented by instances of the `BigPoly` class in SEAL. In order to encrypt integers or rational numbers, one needs to encode them into elements of R_t. SEAL provides a few different encoders for this purpose (see Section 5).

3.2 Ciphertext Space

Ciphertexts in FV are vectors of polynomials in R_q. These vectors 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 vectors; homomorphic multiplications are slightly more complicated and will be described below. Ciphertexts are represented by instances of the `BigPolyArray` class in SEAL.
Textbook-FV only allows ciphertexts of size 2, resulting in minor changes to the homomorphic operations compared to their original description in [20]. We will describe below the algorithms that are implemented in SEAL.

3.3 Encryption and Decryption

Ciphertexts in SEAL are encrypted exactly as described in [20]. A SEAL ciphertext $\mathtt{ct} = (c_0, \ldots, c_k)$ is decrypted by computing

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

Encryption are decryption are implemented in SEAL by the `Encryptor` and `Decryptor` classes, respectively.

3.4 Addition

Suppose two SEAL ciphertexts $\mathtt{ct}_1 = (c_0, \ldots, c_j)$ and $\mathtt{ct}_2 = (d_0, \ldots d_k)$ encrypt plaintext polynomials m_1 and m_2, respectively. Suppose WLOG $j \leq k$. Then

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

encrypts $[m_1 + m_2]_t$.
In SEAL homomorphic addition is implemented as `Evaluator::add`. Similarly, homomorphic subtraction is implemented as `Evaluator::sub`.

3.5 Multiplication

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 ciphertexts of sizes $j+1$ and $k+1$, respectively. The output of $\mathtt{Mul}(\mathtt{ct}_1, \mathtt{ct}_2)$ is a ciphertext $\mathtt{ct}_{\text{mult}} = (C_0, C_1, \ldots, C_{j+k})$ of size $j+k+1$. 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 we define the function `Mul` (or rather family of functions) to mean this generalization of the Textbook-FV multiplication operation (without relinearization). It is implemented as `Evaluator::multiply`.

Algorithms for polynomial multiplication. Multiplication of polynomials in $\mathbb{Z}[x]/(x^n+1)$ is the most computationally expensive part of `Mul`, which in SEAL we implement using Nussbaumer convolution [16]. Note that here polynomial multiplication needs to be performed with integer coefficients, whereas in other homomorphic operations it is done modulo q, which is significantly easier, and can always be done more efficiently using the Number Theoretic Transform (NTT).
It is also possible to implement a Karatsuba-like trick to reduce the number of calls to Nussbaumer convolution, reducing the number of polynomial multiplications to multiply two ciphertexts of sizes k_1 and k_2 from $k_1 k_2$ to ck_1k_2, where $c \in (0,1)$ is some constant depending on k_1 and k_2. For example, if $k_1 = k_2 = 2$, then $c = 3/4$, which is currently the only case implemented in SEAL.

3.6 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 3.5. In other words, given a size $k+1$ ciphertext $(c_0, \ldots, c_k)$ that can be decrypted as was shown in Section 3.3, 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.
Let w denote a power of 2, 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. Now consider the `EvaluateKeyGen` (version 1) algorithm in [20], which for every $i \in \{0, \ldots, \ell\}$ samples $a_i \xleftarrow{\$} R_q$, $e_i \leftarrow \chi$, and outputs the vector

$$\mathtt{evk}_2 = \left[\left([-(a_0 s + e_0) + w^0 s^2]_q, a_0 \right), \ldots, \left([-(a_\ell s + e_\ell) + w^\ell s^2]_q, a_\ell \right) \right] .$$

In SEAL we generalize this to j-power evaluation keys by sampling several a_i and e_i as above, and setting instead

$$\mathtt{evk}_j = \left[\left([-(a_0 s + e_0) + w^0 s^j]_q, a_0 \right), \ldots, \left([-(a_\ell s + e_\ell) + w^\ell s^j]_q, a_\ell \right) \right] .$$

Suppose we have a set of evaluation keys $\mathtt{evk}_2, \ldots, \mathtt{evk}_k$. Then relinearization converts $(c_0, c_1, \ldots, c_k)$ into $(c'_0, c'_1, \ldots, c'_{k-1})$, where

$$c'_0 = c_0 + \sum_{i=0}^{\ell} \mathtt{evk}_k[i][0] c_k^{(i)}, \quad c'_1 = c_1 + \sum_{i=0}^{\ell} \mathtt{evk}_k[i][1] c_k^{(i)} ,$$

and $c'_j = c_j$ for $2 \le j \le k-1$.
Note that in order to generate evaluation keys access to the secret key is needed. This means that the owner of the secret key must generate an appropriate number of evaluation keys and share them with the evaluating party in advance of the relinearization computation, which further

means that the evaluating party needs to inform the owner of the secret key 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 $\texttt{evk}_2$ will be needed. SEAL implements the above operation as `Evaluator::relinearize`.

3.7 Other Homomorphic Operations

In addition to the operations described above, SEAL implements a few other useful operations, such as negation (`Evaluator::negate`), multiplication by a plaintext polynomial (`Evaluator::multiply_plain`), addition (`Evaluator::add_plain`) and subtraction (`Evaluator::sub_plain`) of a plaintext polynomial, noise-optimal product of several ciphertexts (`Evaluator::multiply_many`), exponentiation with relinearization at every step (`Evaluator:exponentiate`), and a sum of several ciphertexts (`Evaluator::add_many`).
SEAL has a fast algorithm for computing the product of a ciphertext with itself. 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. This is implemented as `Evaluator::square`.

3.8 Key Distribution

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 we instead sample the key uniformly from R_3, i.e. we use coefficients $\{-1, 0, 1\}$.

4 Encryption Parameters

Everything in SEAL starts with the construction of an instance of a container that holds the encryption parameters (`EncryptionParameters`). These parameters are:

- `poly_modulus`: a polynomial $x^n + 1$;
- `coeff_modulus`: an integer modulus q;
- `plain_modulus`: an integer modulus t;
- `noise_standard_deviation`: a standard deviation σ;
- `noise_max_deviation`: a bound for the error distribution B;
- `decomposition_bit_count`: the logarithm $\log w$ of w (Section 3.6);
- `random_generator`: a source of randomness.

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 the decomposition bit count is not set (to a non-zero value), SEAL will assume that no relinearization is going to be performed, and prevents the creation of any evaluation keys. If no randomness source is given, SEAL will automatically use `std::random_device`.
In this section we will describe the encryption parameters and their impact on performance. We will discuss security in Section 7. In Section 4.4 we will discuss the automatic parameter selection tools in SEAL, which can assist the user in determining (close to) optimal encryption parameters for many types of computations.

4.1 Default Values

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. As we also mentioned above, unless they want to use relinearization, the user does not need to set `decomposition_bit_count`. By default the constructor will set its value to zero, which will prevent the construction of evaluation keys.
SEAL comes with a list of pairs (n, q) that are returned by the static function

`ChooserEvaluator::default_parameter_options()`

as a keyed list (`std::map`). The default (n, q) pairs are presented in Table 2.

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

Table 2: Default pairs (n, q).

4.2 Polynomial Modulus

The polynomial modulus (`poly_modulus`) is required to 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 7).
Using a larger n decreases performance. On the other hand, it 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. A large value of t allows the scheme to support larger integer arithmetic. When CRT batching is used (Section 5.3), a larger n will allow for more elements of $\mathbb{Z}_t$ to be batched into one plaintext.

4.3 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[4] (see Section 6), and the security level[5] (see Section 7.2 and references therein).
In principle we can take q to be any integer, but taking q to be of special form provides performance benefits. 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.
Second, if q is a prime with $2n|(q-1)$, then SEAL can use the Number Theoretic Transform (NTT) for polynomial multiplications, resulting in huge performance benefits in encryption, relinearization and decryption. SEAL uses David Harvey's algorithm for NTT, as described in [27], which additionally requires that $4q \leq \beta$, where β denotes the *word size* of q:

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

Third, if $t|(q-1)$ (i.e. $r_t(q) = 1$), then the noise growth properties are improved in certain homomorphic operations (recall Table 3).
The default parameters in Table 2 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 when t is a power of 2 up to 2^{18}.
We note that when using CRT batching (see Section 5.3) it will not be possible to have t be a power of 2, as t needs to instead be a prime of a 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.

4.4 Automatic Parameter Selection

To assist the user in choosing parameters for a specific computation, SEAL provides an automatic parameter selection tool. It consists of two parts: a `Simulator` component that simulates noise growth in homomorphic operations using the estimates of Table 3, 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.

5 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 that plaintext elements in the FV scheme are polynomials in R_t. In typical applications of homomorphic encryption, the user would instead want to perform computations on integers or rational numbers.

[4] Bigger q means higher noise bound (good).
[5] Bigger q means lower security (bad).

Encoders are responsible for converting the user's inputs to polynomials in R_t by applying an encoding map. In order for the operations on ciphertexts to reflect the operations on the inputs, the encoding and decoding maps need to respect addition and multiplication.

5.1 Integer Encoder

In SEAL the *integer encoder* is used to encode integers into plaintext polynomials. Despite its name, the integer encoder is really a *family* of encoders, one for each integer base $\beta \geq 2$.
When $\beta = 2$, the idea of the integer encoder is to encode an integer a in the range $[-(2^n - 1), 2^n - 1]$ as follows. It forms the (up to n-bit) binary expansion of $|a|$, say $a_{n-1} \ldots a_1 a_0$, and outputs the polynomial

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

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

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

When β is set to some integer larger than 2, instead of a binary expansion (as was done in the example above) a base-β expansion is used. SEAL uses a *balanced* base-β representation to keep the absolute values of the coefficients as small as possible [19].
Note that the infinity norm of a freshly encoded plaintext polynomial is bounded by $\beta/2$, and the degree of the polynomial encoding a is bounded by $\lceil \log_\beta(|a|) \rceil$. However, as homomorphic operations are performed on the encryptions, the infinity norm and degree will both grow. When the degree becomes greater than or equal to n, or the infinity norm greater than $t/2$, the polynomial will "wrap around" in R_t, yielding an incorrect result. In order to get the correct result, one needs to choose n and t to accommodate the largest plaintext polynomial appearing during the computation. For a very nice estimate on how large n and t need to be, we refer the reader to [15].
The integer encoder is available in SEAL through the `IntegerEncoder` class. Its constructor will require both the `plain_modulus` and the base β as parameters. If no base is given, the default value $\beta = 2$ is used.

5.2 Fractional Encoder

There are several ways for encoding rational numbers in SEAL. One way is to simply scale all rational numbers to integers, encode them using the integer encoder described above, and record the scaling factor in the clear as a part of the ciphertext. We then need to keep track of the scaling during computations, which results in some inefficiency. Here we describe what we call the *fractional encoder*, which has the benefit of automatically keeping track of the scaling. Just like the integer encoder,

the fractional encoder is really a family of encoders, parametrized by an integer base $\beta \geq 2$. The function of this base is exactly the same as in the integer encoder, and we will only explain how the fractional encoder works when $\beta = 2$.
Consider the rational number 5.8125, with the 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 $\texttt{IntegerEncode}(5, \beta = 2) = x^2 + 1$. Then we take the fractional part, add n (degree of the polynomial modulus) to each exponent, and convert it into a polynomial by changing the base 2 into the variable x. Finally we flip the signs of each of the terms, in this case obtaining $-x^{n-1} - x^{n-2} - x^{n-4}$. This defines $\texttt{FracEncode}(r, \beta = 2)$ for rational numbers $r \in [0, 1)$. For any rational number r with a finite binary expansion, we set

$$\begin{aligned}\texttt{FracEncode}(r, \beta = 2) = \operatorname{sign}(r) \cdot [&\texttt{IntegerEncode}(\lfloor |r| \rfloor, \beta = 2) \\ &+ \texttt{FracEncode}(\{|r|\}, \beta = 2)]\,,\end{aligned}$$

where the fractional part is denoted by $\{\cdot\}$. Concluding our example, $\texttt{FracEncode}(5.8125, \beta = 2)$ yields the polynomial $-x^{n-1} - x^{n-2} - x^{n-4} + x^2 + 1$. Decoding works by reversing the steps described above. It is easy to see that `FracEncode` respects both addition and multiplication [19].
The fractional encoder is implemented by the class `FractionalEncoder`. Its constructor will take as parameters the `plain_modulus`, the base β, and positive integers n_f and n_i with $n_f + n_i \leq n$, which describe how many coefficients are reserved for the fractional and integer parts, respectively.[6] If no base is given, the default value $\beta = 2$ is used.

Comparing the two fractional encoding approaches. The *scale-to-integer* technique mentioned above, and our fractional encoder, have similar performance and limitations, but are not equivalent. In some cases the fractional encoder is strictly better.
For example, suppose the homomorphic operations result in some cancellations in the underlying plaintext. Since the level of a scaled encoder never drops, it does not recognize this cancellation, and once the level reaches its maximum (n coefficients), decoding will fail. For the fractional encoder, however, cancellations take care of themselves, permitting potentially more homomorphic operations. As a concrete example, consider $n = 8$, base $\beta = 2$, and the computation $(12 \cdot 0.25)^3$. With the scale-to-integer technique, a rational number $a/2^i$ is encoded as $(p(x), i)$, where $p(x)$ is an integer encoding of a. Hence, the inputs are encoded as $(x^3 + x^2, 0)$, and $(0, 2)$. The result of the computation is $(3x^7 + x^6 - x - 3, 6)$, which does not decode to the correct result since the first entry wrapped around $x^n + 1$. On the other hand, with the fractional encoder, the two inputs are encoded as $x^3 + x^2$ and $-x^6$, and the resulting plaintext polynomial is equal to $(x + 1)^3$, which decodes correctly.

[6] More precisely, n_f describes how many coefficients are used when truncating possibly infinite base-β expansions of rational numbers.

Remark 1. In [15] the authors claimed that the two fractional encoding methods above are equivalent, by claiming the existence of an isomorphism between the underlying rings. We would like to point out that their object R_1 does not satisfy the distribution law, hence is not a ring. This was likely an innocent typo (indeed, with a sign mistake fixed R_1 does become a ring), but even then the map $\phi : R_1 \to R_2$ in their paper is only a surjective homomorphism, and not injective, due to the fact that encoding is not unique: e.g. (x^i, i) encodes the integer 1 for all i.

5.3 CRT Batching

The *CRT (Chinese Remainder Theorem) batching* technique allows up to n integers modulo t to be packed into one plaintext polynomial, and operating on those integers in a *SIMD (Single Instruction, Multiple Data)* manner. For more details and applications we refer the reader to [11, 36, 19].

Batching provides the maximal number of plaintext slots 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. Then there exists (see e.g. [19]) a ring isomorphism $\texttt{Decompose} : R_t \to \prod_{i=0}^{n-1} \mathbb{Z}_t$, whose inverse we denote by `Compose`. In SEAL, `Compose` and `Decompose` are computed using a negacyclic variant of the Number Theoretic Transform (NTT).

When used correctly, batching can provide an enormous performance improvement over the other encoders. Note, however, that for computations on encrypted integers rather than on integers modulo t one needs to ensure that the values in the individual *slots* never wrap around t during the computation.

SEAL provides all of the batching-related tools in the `PolyCRTBuilder` class.

6 Inherent Noise

Definition 1 (Inherent noise). *Let* $\texttt{ct} = (c_0, c_1, \ldots, c_k)$ *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*

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

for some polynomial a*.*

It is proved in [20], that the function (or family of functions) `Decrypt`, as presented in Section 3.3, correctly decrypts a ciphertext as long as the inherent noise satisfies $\|v\| < \Delta/2$.

6.1 Overview of Noise Growth

We present in Table 3 probabilistic estimates of noise growth in some of the most common homomorphic operations. Even though these are estimates, they are simple and work well in practice. For input ciphertexts $\texttt{ct}_i$ we denote their respective inherent noises by v_i. When there is a single encrypted input `ct` we denote its inherent noise by v.

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)`
`Relinearize`	Ciphertext `ct` of size K and target size $L < K$	$\|v\|$ $+(K-L)\sqrt{n}B(\ell+1)w$

Table 3: Noise estimates for homomorphic operations in SEAL.

6.2 Maximal Levels for Default Parameters

In Table 4 we give the maximal supported levels for various power-of-2 plaintext moduli, only taking the noise growth into account. The coefficient moduli are chosen to be the defaults, given in Table 2. We chose to use a uniformly random polynomial in R_t as the plaintext.

n	$\log_2 q$	$\log_2 t$	Max. level
2^{10}	35	6	1
2^{11}	60	7	2
		16	1
2^{12}	116	1	6
		8	4
		20	2
2^{13}	226	8	8
		20	5
		30	3
2^{14}	435	8	15
		32	7
		64	4

Table 4: Maximal levels for different choices of polynomial modulus and plaintext modulus.

7 Security of FV

7.1 Ring-Learning With Errors

The security of the FV encryption scheme is based on the apparent hardness of the famous *Ring-Learning with Errors (RLWE)* problem [30]. Each RLWE sample can be used to extract n *Learning with Errors (LWE)* samples [34, 28]. The concrete hardness depends on the parameters n, q, and the standard deviation of the error distribution σ.

7.2 Security of the Default Parameters in SEAL v2.1

We now give an estimate of the security of the default parameters in SEAL v2.1 based on the LWE estimator of [7].[7] The estimator takes as input an LWE instance given by a dimension n, a modulus q, and a *relative error* $\alpha = \sqrt{2\pi}\sigma/q$. For various attacks it returns estimates for the number of bit operations, memory, and number of samples required to break the LWE instance. In Table 5 we give the expected number of bit operations required to attack the LWE instances induced by the SEAL v2.1 default parameters, assuming that the attacker has as many samples, and as much memory, as they would require. Recall from Section 4.1 that in SEAL the default standard deviation is $\sigma = 3.19$, so we always have $\alpha q = \sigma\sqrt{2\pi} \approx 8$, and we use $\alpha = 8/q$. We use the default n and q as presented in Table 2.

n	q	α	small sis	bkw	sis	dec	kannan
1024	$2^{35} - 2^{14} + 2^{11} + 1$	$8/q$	97.6	237.4	126.5	116.1	116.6
2048	$2^{60} - 2^{14} + 1$	$8/q$	115.1	391.2	136.2	129.0	129.5
4096	$2^{116} - 2^{18} + 1$	$8/q$	119.1	615.3	132.7	128.2	129.2
8192	$2^{226} - 2^{26} + 1$	$8/q$	123.1	1168.6	132.2	—	131.1
16384	$2^{435} - 2^{33} + 1$	$8/q$	130.5	1783.5	134.4	—	135.9

Table 5: Estimates of log *of the bit operations required to perform the above named attacks on the SEAL v2.1 default parameters. The symbol '—' denotes that the estimator did not return a result.*

Recently, Albrecht [3] described new attacks on LWE instances where the secret is very small, and presented estimates of the cost of these attacks on the default parameters used in SEAL v2.0. Estimates for cost of the attacks described in [3] have been included into the LWE estimator of [7]. In Table 5 we have included the attack presented in [3, Sections 3 and 4], labelled 'small sis', which performs best against the SEAL v2.1 parameters. To label the other attacks we follow the notation of [7]: 'bkw' denotes a variant [23] of the BKW attack [10, 5], 'sis' denotes a distinguishing attack as described in [32]; 'dec' denotes a decoding attack as described in e.g. [29]; 'kannan' denotes the attack described in [6]. The

[7] We used the version available on February 23rd, 2017 (commit `d70e1e9`).

estimator was not run for Arora-Ge type attacks [8, 4] or for meet-in-the-middle type attacks, since these are both expected to be very costly.

Remark 2. At the time of writing this, determining the concrete hardness of parametrizations of (R)LWE is an active area of research (see e.g. [17, 13, 7]), and no standardized (R)LWE parameter sets exist. Therefore, when using SEAL or any other implementation of (R)LWE-based cryptography, we strongly recommend the user to consult experts in the security of (R)LWE when choosing which parameters to use.

References

[1] FV-NFLlib. `https://github.com/CryptoExperts/FV-NFLlib`. Accessed: 2017-02-17.

[2] HElib. `https://github.com/shaih/HElib`. Accessed: 2016-11-21.

[3] Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. Cryptology ePrint Archive, Report 2017/047, 2017. `http://eprint.iacr.org/2017/047`.

[4] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic algorithms for LWE problems. *IACR Cryptology ePrint Archive*, 2014:1018, 2014.

[5] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the BKW algorithm on LWE. *Des. Codes Cryptography*, 74(2):325–354, 2015.

[6] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-svp. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013.

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

[8] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[9] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. Cryptology ePrint Archive, Report 2016/510, 2016. `http://eprint.iacr.org/2016/510`.

[10] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003.

[11] 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.

[12] 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.

[13] 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.

[14] Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In Sako [35], pages 325–340.

[15] Anamaria Costache, Nigel P Smart, Srinivas Vivek, and Adrian Waller. Fixed point arithmetic in SHE schemes. Technical report, Cryptology ePrint Archive, Report 2016/250, 2016. http://eprint.iacr. org/2016/250.

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

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

[18] Eric Crockett and Chris Peikert. *Λoλ*: Functional lattice cryptography. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 993–1005. ACM, 2016.

[19] 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`.

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

[21] Rosario Gennaro and Matthew Robshaw, editors. *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*. Springer, 2015.

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

[23] Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-bkw: Solving LWE using lattice codes. In Gennaro and Robshaw [21], pages 23–42.

[24] Shai Halevi and Victor Shoup. Design and Implementation of a Homomorphic-Encryption Library. `http://people.csail.mit.edu/shaih/pubs/he-library.pdf`, 2013.

[25] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.

[26] Shai Halevi and Victor Shoup. Bootstrapping for helib. In Oswald and Fischlin [33], pages 641–670.

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

[28] 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.

[29] 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.

[30] 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.

[31] Carlos Aguilar Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrède Lepoint. Nfllib: Ntt-based fast lattice library. In Sako [35], pages 341–356.

[32] Daniele Micciancio and Oded Regev. *Lattice-based Cryptography*, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[33] Elisabeth Oswald and Marc Fischlin, editors. *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*. Springer, 2015.

[34] 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.

[35] Kazue Sako, editor. *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, volume 9610 of *Lecture Notes in Computer Science*. Springer, 2016.

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