Simple Encrypted Arithmetic Library 2.3.1

Kim Laine

Microsoft Research, WA, USA; kim.laine@microsoft.com

1 Introduction

Traditional encryption schemes, both symmetric and asymmetric, were not designed to respect any algebraic structure of the plaintext and ciphertext spaces, i.e. no computations can be performed on the ciphertext in a way that would pass through the encryption to the underlying plaintext without using the secret key, and such a property would in many contexts be considered a vulnerability. Nevertheless, this property has powerful applications, e.g. in outsourced (cloud) computation scenarios the cloud provider could use this to guarantee customer data privacy in the presence of both internal (malicious employee) and external (outside attacker) threats. An encryption scheme that allows computations to be done directly on the encrypted data is said to be a *homomorphic encryption scheme*.

Some schemes, such as ElGamal (resp. e.g. Paillier), are multiplicatively homomorphic (resp. additively homomorphic), i.e. one algebraic operation can pass through the encryption to the underlying plaintext data. The restriction to one single operation is very strong, and instead a much more powerful *fully* homomorphic encryption scheme that respects both additions and multiplications would be needed for many interesting applications, as it would allow arbitrary Boolean or arithmetic circuits to be evaluated. The first such encryption scheme was invented by Craig Gentry in 2009 [22], and since then researchers have introduced a number of new and more efficient fully homomorphic encryption schemes [11, 10, 7, 9, 21, 29, 5, 24, 15].

Despite the promising theoretical power of homomorphic encryption, the practical side remained underdeveloped for a long time. Recently new implementations, new data encoding techniques, and new applications have started to improve the situation, but much remains to be done. In 2015 the first version of the *Simple Encrypted Arithmetic Library* SEAL was released, with the specific 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.

This documents describes the core features of SEAL 2.3.1, and attempts to provide a practical high-level guide to using homomorphic encryption for a wide audience. For a more hands-on experience we recommend the reader to go over the code examples that come with the library, and to read through the detailed comments accompanying the examples. This is particularly important for users of previous versions of SEAL.

The library is available through `http://sealcrypto.org`, and is licensed under the MSR License Agreement. For the license, see `LICENSE.txt` distributed with the code. This document refers to SEAL 2.3.1.

1.1 Roadmap

In Section 2 we give an overview of changes moving from SEAL v2.3.0-4 to SEAL 2.3.1, which are expanded upon in the other sections of this document. In Section 3 we define notation and parameters that are used throughout this document. In Section 4 we give the description of the Brakerski/Fan-Vercauteren homomorphic encryption scheme (BFV) – as originally specified in [21] – and in Section 5 we describe how SEAL 2.3.1 differs from this original description. In Section 6 we introduce the new notion of ciphertext noise and we discuss the expected noise growth behavior of SEAL ciphertexts as homomorphic evaluations are performed. In Section 7 we discuss the available ways of encoding data into SEAL 2.3.1 plaintexts. In Section 8 we discuss the selection of parameters for performance, and describe the automatic parameter selection module. In Section 9 we discuss the security properties of SEAL 2.3.1.

1.2 Acknowledgments

This document builds on previous versions that have been contributed to by Hao Chen, Rachel Player, Amir Jalali, Zhicong Huang, and Kyoohyung Han.

2 Overview of Changes in SEAL 2.3.1

2.1 Updated Default Parameters

The default encryption parameters in SEAL have been updated according to the most recent draft of the homomorphic encryption security standard by the HomomorphicEncryption.org group (see [13] for previous version of the document). In particular, parameters for a 256-bit security level have been added. See Table 3 for the new parameters.

2.2 Improved Linux and OS X support

SEAL now compiles with both `clang++-5` and `g++-8`. Building and installation is easy using CMake. To configure, change to the SEAL directory and run

```
cmake .
```

For easier access to the configuration options use e.g. `ccmake` instead of `cmake`. The configuration options allow compiling in either Release or Debug mode (very slow), enabling/disabling the use of specific compiler intrinsics, or enabling/disabling Microsoft GSL support (see Section 2.5). Next run

```
make && sudo make install
```

to build and install SEAL system-wide. To install locally, set the `CMAKE_INSTALL_PREFIX` to a desired value in e.g. `ccmake`.

Using SEAL in applications is now very easy. In a `CMakeLists.txt` file for your applications include the line

```
find_package(SEAL 2.3.1 EXACT REQUIRED)
```

This will import the target `SEAL::seal`. Simply link `SEAL::seal` with your program or library and everything should work.

2.3 Generic Galois Automorphisms

SEAL now allows generic Galois automorphisms for permuting the values in the slots. These are exposed through the `Evaluator::apply_galois` function.

2.4 Changes to Memory Pool

Thread-unsafe memory pools are no longer available by default through the `MemoryPoolHandle` class for safety reasons.

Tool classes such as `KeyGenerator` and `Evaluator`, etc. no longer take `MemoryPoolHandle` as a constructor argument. These caused a lot of confusion, and detracted from the much more important practice of using thread-local `MemoryPoolHandle` arguments to member function calls such as `Evaluator::multiply`.

2.5 Support for Microsoft GSL (experimental)

SEAL can use Microsoft GSL (MSGSL)[1] classes `gsl::span` and `gsl::multi_span` for providing more convenient hierarchical array access to objects such as `Ciphertext`, and for passing matrix/vector arguments to `PolyCRTBuilder::compose` and `PolyCRTBuilder::decompose` without requiring the allocation to necessarily be in the form of an `std::vector`.

MSGSL support is experimental at this point, and hence disabled by default. To enable it in Microsoft Visual Studio, simply download and install MSGSL and add the installation path to SEAL include directories in the project settings. When using CMake, first download and install MSGSL; then enable `SEAL_USE_MSGSL` in `ccmake`. If CMake fails to find your installation of MSGSL, you will need to add the path to `CMAKE_INCLUDE_PATH` in `ccmake`.

If SEAL is compiled and installed with MSGSL support,

```
find_package(SEAL 2.3.1 EXACT REQUIRED)
```

will additionally import a target `SEAL::msgsl`, which will automatically be linked with your program or library when you link the target `SEAL::seal`. Thus, no additional changes to your own `CMakeLists.txt` are needed.

2.6 Other

According to the practices adopted by the HomomorphicEncryption.org group, the encryption scheme implemented in SEAL is now called the Brakerski/Fan-Vercauteren scheme (BFV), as opposed to the previously used name Fan-Vercauteren scheme (FV). This is only a change in the name and indicates no change in the encryption scheme itself.

The .NET examples project `SEALNETExamples` now contains the same performance test examples that have been available in `SEALExamples`.

In addition to these visible changes, SEAL 2.3.1 brings in many internal improvements and bug fixes.

3 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 operation to each coefficient separately. The norm $\|\cdot\|$ denotes the infinity norm and $\|\cdot\|^{\mathrm{can}}$ denotes the *canonical norm* [16, 23]. 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 2.3.1.

[1] Microsoft GSL is available at `https://github.com/Microsoft/GSL/`.

Parameter	Description	Name in SEAL (if applicable)
q	Modulus in the ciphertext space (coefficient modulus) of the form $q_1 \times \ldots \times q_k$, where q_i are prime	`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.

4 The BFV Scheme

In this section we give the definition of the BFV scheme as presented in [21].

4.1 Plaintext Space and Encodings

In BFV 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 7 and demonstrated in the SEALExamples project that comes with the code.

4.2 Ciphertext Space

Ciphertexts in BFV 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.

4.3 Description of Textbook-BFV

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

- `SecretKeyGen`(λ): Sample $s \xleftarrow{\$} R_2$ and output $\texttt{sk} = s$.
- `PublicKeyGen`($\texttt{sk}$): Set $s = \texttt{sk}$, sample $a \xleftarrow{\$} R_q$, and $e \leftarrow \chi$. Output $\texttt{pk} = ([-(as + e)]_q, a)$.
- `EvaluationKeyGen`($\texttt{sk}, w$): for $i \in \{0, \ldots, \ell\}$, sample $a_i \xleftarrow{\$} R_q$, $e_i \leftarrow \chi$. Output

$$\texttt{evk} = \left([-(a_i s + e_i) + w^i s^2]_q, a_i\right) .$$

- `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').

5 How SEAL Differs from Textbook-BFV

In practice, some operations in SEAL are done slightly differently, or in slightly more generality, than in textbook-BFV. In this section we discuss these differences in detail.

5.1 Plaintexts and Ciphertexts

Plaintext elements in SEAL are polynomials in R_t, just as in textbook-BFV. Ciphertexts in SEAL are tuples of polynomials in R_q of length at least 2. This is a difference to textbook-BFV, where the ciphertexts are always in $R_q \times R_q$.

5.2 Decryption

A SEAL ciphertext $\texttt{ct} = (c_0, \ldots, c_k)$ is decrypted by computing

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

This generalization of decryption (compare to Section 4.3) 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].

5.3 Multiplication

Consider the `Multiply` function as described in Section 4.3. The first step that outputs the intermediate ciphertext (c_0, c_1, c_2) defines a function `Evaluator::multiply`, 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 5.2), 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 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 $\mathtt{ct}_{\mathrm{mult}} = (C_0, C_1, \ldots, C_{j+k})$. 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 the function `Multiply` means this generalization of the first step of multiplication. It is implemented as `Evaluator::multiply`.

5.4 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 5.3. In other words, given a size $k+1$ ciphertext $(c_0, \ldots, c_k)$ that can be decrypted as was shown in Section 5.2, 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 BFV, 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 $\mathtt{evk} = \big([-(as+e)+s^2]_q, a\big)$, where $a \xleftarrow{\$} R_q$, and $e \leftarrow \chi$. Now set $c'_0 = c_0 + \mathtt{evk}[0]c_2$, $c'_1 = c_1 + \mathtt{evk}[1]c_2$, 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 5.2), 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. [11, 9, 7, 21]) 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 4.3. Then one can show that instead setting c'_0 and c'_1 as in Section 4.3 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 $\mathtt{EvaluationKeyGen}(\mathtt{sk}, w)$

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 4.3. 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 $2 \le j \le k-1$.

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 we define the function `Relinearize` to mean this generalization of the second step of multiplication as was described in Section 4.3. It is implemented as `Evaluator::relinearize`. Suppose a ciphertext `ct` has size $K > 2$, and $\texttt{evk} = \{\texttt{evk}_2, \texttt{evk}_3, \ldots, \texttt{evk}_{K-1}\}$ is a set of evaluation keys generated with `KeyGenerator::generate_evaluation_keys` in SEAL, then `relinearize(ct,evk)` returns a ciphertext of size 2 encrypting the same message as `ct`.

5.5 Addition

We also need to generalize addition to be able to operate on ciphertexts of any size. Suppose we have two SEAL 2.3.1 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 $j \le k$. 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 2.3.1 we define the functions `Add` 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`.

5.6 Galois Automorphisms

SEAL 2.3.1 allows the user to apply Galois automorphisms of the cyclotomic extension $\mathbb{Q} \hookrightarrow \mathbb{Q}[x]/(x^n + 1)$, where $x^n + 1$ is the polynomial modulus, to the plaintext polynomials in encrypted form. We will not discuss the details of what this means here, and instead refer the user to any introductory text on algebraic number theory. Simply put, the extension is generated by any primitive $m = 2n$-th root of unity. If ζ is such a primitive root, then the other primitive roots are $\zeta^3, \zeta^5, \ldots, \zeta^{m-1}$. The Galois automorphisms correspond to changing the primitive root as $\zeta \mapsto \zeta^{2k-1}$, and in the cyclotomic extension ring corresponds to sending a polynomial $f(x) \mapsto f(x^{2k-1})$. Restricting to $\mathbb{Z}[x]/(x^n+1)$ and reducing coefficients modulo the plaintext modulus t yields a corresponding operation apply_galois(ct, gal_elt, gal_keys) in the plaintext space. Here gal_elt is the *Galois element* that determines the Galois automorphism; this is the odd exponent $2k-1$ above. gal_keys denotes *Galois keys*—a special type of key required by the Galois automorphism operation. Galois keys for a specific Galois element can be generated with the KeyGenerator::generate_galois_keys function, and apply_galois is implemented as Evaluator::apply_galois. There is a special overload of KeyGenerator::generate_galois_keys that generates Galois keys for logarithmically many (in n) Galois automorphisms that can be used for apply_galois with and gal_elt.

The Galois automorphisms form a group (under composition), which is isomorphic to $\mathbb{Z}_{n/2} \times \mathbb{Z}_2$. The first factor is generated by gal_elt $= 3$, and the second factor is generated by gal_elt $= m - 1$. This is important, because in the batching view (see Section 7.4) where $\zeta \in \mathbb{Z}_t^*$, the plaintext can be viewed as a $2 \times (n/2)$ matrix whose rows and columns can be cyclically rotated by applying the corresponding Galois automorphisms. These operations are implemented as Evaluator::rotate_rows and Evaluator::rotate_columns.

5.7 Other Operations

SEAL provides a function Negate to perform homomorphic negation. This is implemented in the library as Evaluator::negate.

SEAL provides functions AddPlain(ct, m_{add}) and MultiplyPlain(ct, m_{mult}) that, given a ciphertext ct encrypting a plaintext polynomial m, and unencrypted plaintext polynomials $m_{\mathrm{add}}, m_{\mathrm{mult}}$, output encryptions of $m + m_{\mathrm{add}}$ and $m \cdot m_{\mathrm{mult}}$, 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. The 'plain' operations are implemented in SEAL 2.3.1 as Evaluator::add_plain and Evaluator::multiply_plain. Analogously to AddPlain we have implemented a plaintext subtraction function 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 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. Evaluator::multiply_many relinearizes after every multiplication it performs, which means that the user needs to provide it an appropriate set of evaluation keys as input.

SEAL 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.

With parameter sets that support the Number Theoretic Transform (NTT) (see Section 8.5 and Section 8.6), Evaluator::multiply_plain works by first applying the Number Theoretic

Transform (NTT) to both the input ciphertext, and the input plaintext, then performing a dyadic product of the transformed polynomials, and finally transforming the resulting ciphertext back. In cases where the same input plaintext or ciphertext needs to be used repeatedly for several different plain multiplications, it does not make sense to repeat the transform every single time. Instead, SEAL allows plaintexts and the ciphertexts to be NTT transformed at any time using the functions `Evaluator::transform_to_ntt`. Ciphertexts also can be transformed back from NTT using `Evaluator::transform_from_ntt`. Given a ciphertext and plaintext, both in NTT transformed form, the user can call `Evaluator::multiply_plain_ntt` to perform a very fast plain multiplication operation. The result will still be in NTT transformed form, and can be transformed back with `Evaluator::transform_from_ntt`.

5.8 Composite Coefficient Modulus

The coefficient modulus in SEAL is composed of several distinct prime values. In particular, all the homomorphic operations over the polynomial coefficients ring is implemented based on *residue number system* (RNS) arithmetic. We adopt several optimization techniques in low level arithmetic implementation which improve the performance significantly, as proposed in [4]. Here we describe this idea briefly at a high level.

Since the core operations of the FV scheme are performed in the polynomial ring R_q for a modulus q, there is no restriction in choosing q to be a product of several distinct prime moduli $q_1, q_2, \ldots, q_k$. The *Chinese Remainder Theorem* (CRT) implies a ring isomorphism $R_q \equiv R_{q_1} \times \ldots \times R_{q_k}$, which means that ring operations can just as well be performed in the factors R_{q_i} separately. Unfortunately, homomorphic multiplication and decryption require more than simply ring operations, most importantly division and rounding. The main contribution of [4] is to show how these operations can nevertheless be performed.

In SEAL 2.3.1, the coefficient modulus is implemented as a vector of `SmallModulus` elements with arbitrary bit-length up to 60-bit. The product of these small moduli constructs the encryption coefficient modulus. We describe the restrictions on these moduli further in Section 8.

SEAL 2.3.1 implements a combination of the classical relinearization operation and the *FullRNS relinearization* described in [4]. As a result, the decomposition bit count can be at most 60. This also applies to Galois automorphisms (Galois keys).

5.9 Key Distribution

In Section 5.4 we already explained how key generation in SEAL 2.3.1 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 $\{0, 1\}$. SEAL instead samples the key uniformly from R_3, i.e. with coefficients in $\{-1, 0, 1\}$.

6 Noise

In this section we present a heuristic noise growth analysis for SEAL. Although in textbook-BFV all ciphertexts have size 2, we allow ciphertexts of any size greater than or equal to 2, and present general results accordingly. SEAL implements the method of [4] which has slightly different noise growth properties than textbook-BFV, but these differences are small and in practice have no effect. Thus, we only analyze textbook-BFV with the arbitrary size ciphertext extension as mentioned above.

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

$$\frac{t}{q}\mathtt{ct}(s) = \frac{t}{q}\left(c_0 + c_1 s + \cdots + c_k s^k\right) = m + v + at \in R \otimes \mathbb{Q}\,,$$

for some polynomial a *with integer coefficients.*

Intuitively, invariant noise captures the notion that the noise v being rounded incorrectly is what causes decryption failures in the BFV scheme. We see this in the following Lemma, which bounds the coefficients of v.

Lemma 1. *The function* `Decrypt`*, as presented in Section 5.2, correctly decrypts a ciphertext* `ct` *encrypting a message* m*, as long as the invariant noise* v *satisfies* $\|v\| < 1/2$*.*

Proof. Let $\mathtt{ct} = (c_0, c_1, \dots, c_k)$. Using the formula for decryption, we have for some polynomial A with integer coefficients:

$$\begin{aligned} m' &= \left[\left\lfloor \frac{t}{q}\left[c_0 + c_1 s + \cdots + c_k s^k\right]_q \right\rceil\right]_t \\ &= \left[\left\lfloor \frac{t}{q}\left(c_0 + c_1 s + \cdots + c_k s^k + Aq\right) \right\rceil\right]_t \\ &= \left[\left\lfloor \frac{t}{q}\left(c_0 + c_1 s + \cdots + c_k s^k\right) + At \right\rceil\right]_t \\ &= \left[\left\lfloor \frac{t}{q}\left(c_0 + c_1 s + \cdots + c_k s^k\right) \right\rceil\right]_t . \end{aligned}$$

Then by definition of invariant noise,

$$m' = [\lfloor m + v + at \rceil]_t = m + \lfloor v \rceil\,.$$

Hence decryption is successful as long as v is removed by the rounding, i.e. if $\|v\| < 1/2$. □

It is often in practice more convenient to talk about how much noise we have left until decryption will fail. We call this the (invariant) *noise budget.*

Definition 2 (Noise budget). *Let* v *be the invariant noise of a ciphertext* `ct` *encrypting the message* $m \in R_t$*. Then the noise budget of* `ct` *is* $-\log_2(2\|v\|)$*.*

Lemma 2. *The function* `Decrypt`*, as presented in Section 5.2, correctly decrypts a ciphertext* `ct` *encrypting a message* m*, as long as the noise budget of* `ct` *is positive.* □

In SEAL the user can output the noise budget in a particular ciphertext using the function `Decryptor::invariant_noise_budget`. Note that this will require having access to the secret key. Users without access to the secret key can instead use the noise simulator (see Section 8.7) to estimate the noise.

6.1 Heuristic Estimates for Noise Growth

Homomorphic operations increase the invariant noise in complicated ways. The reader can find strict upper bounds for the noise growth in the Appendix, along with proofs, but these bounds result in poor practical estimates. SEAL uses instead heuristic upper-bound estimates that hold with very high probability. Similar estimates have previously been presented in [16], but using yet another definition of noise.

The heuristic upper bounds can be obtained by modifying the proofs of the strict upper bounds in Appendix. The key idea is to use the *canonical norm* $\|\cdot\|^{\mathrm{can}}$ instead of the usual infinity norm $\|\cdot\|$, which has the nice property that for any polynomials a, b,

$$\|a\| \leq \|a\|^{\mathrm{can}} \leq \|a\|_1 \,, \quad \|ab\|^{\mathrm{can}} \leq \|a\|^{\mathrm{can}} \|b\|^{\mathrm{can}} \,.$$

Since the usual (infinity) norm is always bounded from above by the canonical norm, it suffices for correctness to ensure that the canonical norm never reaches $1/2$. For more details on exactly how the canonical norm works, we refer the reader to [16, 23].

Lemma 3 (Initial noise heuristic). *Let $\mathtt{ct}$ be a fresh encryption of a message $m \in R_t$. Let N_m be an upper bound on the number of non-zero terms in the polynomial m. The noise v in $\mathtt{ct}$ satisfies*

$$\|v\|^{can} \leq \frac{r_t(q)}{q}\|m\|N_m + \frac{t}{q}\min\{B, 6\sigma\}\left(4\sqrt{3}n + \sqrt{n}\right) \,,$$

with very high probability.

Lemma 4 (Addition heuristic). *Let $\mathtt{ct}_1$ and $\mathtt{ct}_2$ be two ciphertexts encrypting $m_1, m_2 \in R_t$, and having noises v_1, v_2, respectively. Then the noise v_{add} in their sum $\mathtt{ct}_{add}$ satisfies $\|v_{add}\|^{can} \leq \|v_1\|^{can} + \|v_2\|^{can}$.*

Lemma 5 (Multiplication heuristic). *Let $\mathtt{ct}_1$ be a ciphertext of size j_1+1 encrypting m_1 with noise v_1, and let $\mathtt{ct}_2$ be a ciphertext of size j_2+1 encrypting m_2 with noise v_2. Let N_{m_1} and N_{m_2} be upper bounds on the number of non-zero terms in the polynomials m_1 and m_2, respectively. Then the noise v_{mult} in the product $\mathtt{ct}_{mult}$ satisfies the following bound:*

$$\begin{aligned}\|v_{mult}\|^{can} \leq{} & \left(2\|m_1\|N_{m_1} + t\sqrt{3n}\frac{\left(\sqrt{12n}\right)^{j_1+1} - 1}{\sqrt{12n} - 1}\right)\|v_2\|^{can} \\ & + \left(2\|m_2\|N_{m_2} + t\sqrt{3n}\frac{\left(\sqrt{12n}\right)^{j_2+1} - 1}{\sqrt{12n} - 1}\right)\|v_1\|^{can} \\ & + 3\,\|v_1\|^{can}\,\|v_2\|^{can} + \frac{t\sqrt{3n}}{q}\cdot\frac{\left(\sqrt{12n}\right)^{j_1+j_2+1} - 1}{\sqrt{12n} - 1} \,,\end{aligned}$$

with very high probability.

Lemma 6 (Relinearization heuristic). *Let $\mathtt{ct}$ be a ciphertext of size $M+1$ encrypting m, and having noise v. Let $\mathtt{ct}_{relin}$ of size $N+1$ be the ciphertext encrypting m, obtained by the relinearization of $\mathtt{ct}$, where $2 \leq N+1 < M+1$. Then, the noise v_{relin} in $\mathtt{ct}_{relin}$ can be bounded as*

$$\|v_{relin}\|^{can} \leq \|v\|^{can} + \frac{t}{q}\sqrt{3}\min\{B, 6\sigma\}(M-N)n(\ell+1)w \,,$$

with very high probability.

In SEAL 2.3.1 relinearization always relinearizes a ciphertext down to size $N+1=2$, so $N=1$ always.

Remark 1. It is worth mentioning that while the heuristics for initial noise and relinearization look in fact *worse* than the strict upper bounds (see Appendix), the estimate for multiplication is much tighter in the heuristic, and will quickly yield much better upper bound estimates than the strict formula.

Lemma 7 (Plain multiplication heuristic). *Let $\texttt{ct} = (x_0, \ldots, x_j)$ be a ciphertext encrypting m_1 with noise v, and let m_2 be a plaintext polynomial. Let N_{m_2} be an upper bound on the number of non-zero terms in the polynomial m_2. Let $\texttt{ct}_{pmult}$ denote the ciphertext obtained by plain multiplication of $\texttt{ct}$ with m_2. Then the noise v_{pmult} in $\texttt{ct}_{pmult}$ can be bounded as*

$$\|v_{pmult}\|^{can} \leq N_{m_2} \|m_2\| \, \|v\|^{can} .$$

Lemma 8 (Plain addition heuristic). *Let $\texttt{ct} = (x_0, \ldots, x_j)$ be a ciphertext encrypting m_1 with noise v, and let m_2 be a plaintext polynomial. Let $\texttt{ct}_{padd}$ denote the ciphertext obtained by plain addition of $\texttt{ct}$ with m_2. Then the noise v_{padd} in $\texttt{ct}_{padd}$ can be bounded as*

$$\|v_{padd}\|^{can} \leq \|v\|^{can} + \frac{r_t(q)}{q} N_{m_2} \|m_2\| .$$

6.2 Summary of noise growth

SEAL uses slightly simplified versions of the heuristic estimates presented in Section 6.1, as it is easy to see that certain terms are insignificant for any reasonable set of parameters. For a ciphertext $\texttt{ct}$, with invariant noise v, we denote by $\nu(\texttt{ct})$ an upper bound on $\|v\|^{\text{can}}$. For operations that take only one input ciphertext $\texttt{ct}$, we denote $\nu = \nu(\texttt{ct})$. For operations that take several inputs $\texttt{ct}_1, \ldots, \texttt{ct}_k$, we denote $\nu_k = \nu(\texttt{ct}_k)$. For each operation we describe a bound for the noise in the output in terms of ν, or $\nu_1, \ldots, \nu_k$, and the encryption parameters (recall Table 1).

Some operations, such as `AddPlain` and `MultiplyPlain`, take a plaintext polynomial $m \in R_t$ as input. In these cases the bound ν for the output depends also on the qualities of the plaintext polynomial, in particular the infinity norm $\|m\|$, and an upper bound N_m on the number of non-zero coefficients in the polynomial m.

The noise growth estimates implemented in SEAL are summarized in Table 2.

We also 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. Addition and subtraction have a very small impact on noise.
3. Relinearization increases the noise only by an additive factor. Compare this with multiplication, which increases the noise also by a multiplicative factor. This means, for example, that after a few multiplications have been performed, depending on the decomposition bit count (recall Table 1), the additive factor from relinearization can completely drown into the noise in the input.
4. The decomposition bit count has a significant effect on both performance (recall Section 5.4) 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

Operation	Input description	Noise bound of output
`Encrypt`	Plaintext m	$\frac{r_t(q)}{q}\|m\|N_m + \frac{7nt}{q}\min\{B, 6\sigma\}$
`Negate`	Ciphertext $\mathtt{ct}$	ν
`Add/Sub`	Ciphertexts $\mathtt{ct}_1$ and $\mathtt{ct}_2$	$\nu_1 + \nu_2$
`AddPlain/SubPlain`	Ciphertext $\mathtt{ct}$ and plaintext m	$\nu + \frac{r_t(q)}{q}N_m\|m\|$
`MultiplyPlain`	Ciphertext $\mathtt{ct}$ and plaintext m	$N_m\|m\|\nu$
`Multiply`	Ciphertexts $\mathtt{ct}_1$ and $\mathtt{ct}_2$ of sizes j_1+1 and j_2+1	$t\sqrt{3n}\left[(12n)^{j_1/2}\nu_2 + (12n)^{j_2/2}\nu_1 + (12n)^{(j_1+j_2)/2}\right]$
`Square`	Ciphertext $\mathtt{ct}$ of size j	Same as $\mathtt{Multiply}(\mathtt{ct}, \mathtt{ct})$
`Relinearize`	Ciphertext $\mathtt{ct}$ of size K and target size L, such that $2 \leq L < K$	$\nu + \frac{2t}{q}\min\{B, 6\sigma\}(K-L)n(\ell+1)w$
`AddMany`	Ciphertexts $\mathtt{ct}_1, \ldots, \mathtt{ct}_k$	$\sum_i \nu_i$
`MultiplyMany`	Ciphertexts $\mathtt{ct}_1, \ldots, \mathtt{ct}_k$	Apply `Multiply` in a tree-like manner, and `Relinearize` down to size 2 after every multiplication
`Exponentiate`	Ciphertext $\mathtt{ct}$ and exponent k	Apply `MultiplyMany` to k copies of $\mathtt{ct}$

Table 2: Noise estimates for homomorphic operations in SEAL.

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.

7 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 4 that plaintext elements in the FV scheme are polynomials in R_t, 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 injective maps from that particular subset into R_t. Since no non-trivial subset of $\mathbb{Z}$ is closed under additions and multiplications, 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 in the image of the encoding map.

7.1 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. 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 could possibly significantly reduce the noise budget.

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.

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 scalar encoder is *not* implemented in SEAL 2.3.1 due to its inefficiency, but it can be constructed as a special case of some of the other encoders by choosing their parameters in a certain way. These 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.

7.2 Integer Encoder

In SEAL the *integer encoder* is used to encode integers in a much more efficient manner than what the scalar encoder (Section 7.1) 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 $a_{n-1} \ldots a_1 a_0$. 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 2. SEAL uses only unsigned integer data types, so each coefficient of the polynomial is represented as its smallest positive representative modulo t. For example, the -1 coefficients of the polynomial will be stored as the positive 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 smaller noise growth in homomorphic operations.

The integer encoder is available in SEAL 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.

7.3 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 7.2 above), the fractional encoder is a family of encoders, parametrized by an integer base $B \geq 2$ [18]. 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 (as in 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

$$\texttt{FracEncode}(5.8125, B = 2) \cdot \texttt{FracEncode}(2.25, B = 2)$$

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

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 7.2). 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 through the class `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.

7.4 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 [8, 33].

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[2]. In this case the multiplicative

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

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

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

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 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

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

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`. These isomorphisms are computed using a negacylic variant of the Number Theoretic Transform (NTT).

In SEAL the n-dimensional $\mathbb{Z}_t$-vector that `Compose` and `Decompose` convert to and from a plaintext polynomial can be thought of as a $2 \times (n/2)$ matrix, as we already briefly described in Section 5.6. The benefit is that in this case the `apply_galois` operation has specializations `rotate_rows` and `rotate_columns`, which rotate the matrix rows and columns (swap) cyclically a given number of steps in either direction. If Galois keys corresponding to a particular rotation have been generated and are used, the computational cost of the rotation is essentially the same as that of relinearization. If instead logarithmically many (in n) Galois keys were generated (recall Section 5.6), then rotating k steps in either direction is $\min\{\texttt{HammingWeight}(k), \texttt{HammingWeight}(n/2 - k)\}$ times more expensive. Note that in this case rotating the rows power-of-2 number of steps in either direction is essentially as expensive as a single relinearization.

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 7.1), and could be solved by choosing t to be large enough, which will unfortunately cause large noise growth.

SEAL provides `Compose` and `Decompose` functionality in the `PolyCRTBuilder` class. The constructor of `PolyCRTBuilder` takes an instance of `SEALContext` as argument, and will throw an exception unless the parameters are appropriate, as was described in the beginning of this section. The rotations are implemented as `Evaluator::rotate_rows` and `Evaluator::rotate_columns`, and are similarly only available when the parameters support batching.

8 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 $x^n + 1$; n a power of 2;
- `coeff_modulus`: an integer modulus q which is constructed as a product of multiple distinct primes;
- `plain_modulus`: an integer modulus t;
- `noise_standard_deviation`: a standard deviation σ;
- `random_generator`: a source of randomness.

In most cases the user only needs to set the `poly_modulus`, `coeff_modulus`, and `plain_modulus` parameters. Both `random_generator` and `noise_standard_deviation` have good default values and are in most cases not necessary to set explicitly (see Section 8.3).

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 different parameters and their impact. We will discuss security briefly in Section 9. In Section 8.7 we will discuss the automatic parameter selection tools in SEAL, which can help the user in determining optimal encryption parameters for certain use-cases.

8.1 Setting Parameters

Once an `EncryptionParameters` object has been created, the parameters need to be set. This can be done using functions such as `EncryptionParameters::set_coeff_modulus`. Once all of the critical parameters have been set, the user needs to create an instance of the `SEALContext` class, which automatically evaluates the validity and properties of the parameters, and performs a series of pre-computations on them. The properties of the parameters are stored in an instance of the `EncryptionParameterQualifiers` struct, which we describe below in Section 8.6.

8.2 Hash Block

When any of the encryption parameters (except `random_generator`) is changed, SEAL computes and updates an internally stored SHA-3 hash (*hash block*) of the parameters. The hash is automatically stored by every ciphertext, and all key material created under the given parameters, and is used for fast input validity and compatibility checking. The user can modify the hash block by hand and mutate the ciphertext/key data directly, but this should typically never be done unless absolutely necessary for some advanced use-cases.

8.3 Default Values

If the user does not specify σ (`noise_standard_deviation`), it will be set by the constructor of `EncryptionParameters` to the default value of $3.19 \approx 8/\sqrt{2\pi}$. If no randomness source (`random_generator`) is given, SEAL will automatically use `std::random_device`.

The user will have to select n by setting the polynomial modulus (`EncryptionParameters ::set_poly_modulus`) to a polynomial of the form $x^n + 1$, where n is a power of 2. For ease-of-use, SEAL comes with hard-coded default values for q (`coeff_modulus`) corresponding to various realistic choices of n. These default parameters are included for 128-bit, 192-bit, and 256-bit security levels according to the most recent (unpublished at the time of writing) homomorphic encryption security standard by the HomomorphicEncryption.org group. For an earlier draft standard, see [13]. These default values are presented in Table 3, and can be accessed through the functions `coeff_modulus_128`, `coeff_modulus_192`, and `coeff_modulus_256`. The estimates assume σ to be the default value, and omit issues such as the memory cost of the attacks. In Section 9 we will discuss the security properties of SEAL in a bit more detail.

n	Bit-length of default q		
	128-bit security	192-bit security	256-bit security
1024	27	19	14
2048	54	37	29
4096	109	75	58
8192	218	152	118
16384	438	300	237
32768	881	600	476

Table 3: Default pairs (n, q) for 128-bit, 192-bit, and 256-bit security levels.

8.4 Polynomial Modulus

The polynomial modulus (`poly_modulus`) must be a polynomial of the form $x^n + 1$, where n is a power of 2. This is both for security (see Section 9) and performance reasons. 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 7). 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`.

8.5 Coefficient Modulus and Plaintext Modulus

Suppose the polynomial modulus is held fixed. Then the choice of the coefficient modulus q affects two things: the noise budget in a freshly encrypted ciphertext[3] and the security level[4].

In principle we can take q to be any integer, as long as it is not too large to cause security problems. In SEAL, coefficient modulus q is a product of multiple small primes $q_1 \times \ldots \times q_k$. We adopt a generic algorithm for computing modular arithmetic modulo these small primes. Therefore, taking these small primes to be of special form does not provide any performance improvement. The user is free to choose a set of arbitrary primes regarding their requirements as long as they are at most 60-bit long and $q_i = 1 \pmod{2n}$ for $i \in \{1, 2, \ldots, k\}$. We use David Harvey's algorithm for NTT as described in [26].

In some cases the user might want to use a particular n, but the default coefficient modulus for that n is unnecessarily large. In these cases it might be beneficial from the point of view of performance to simply use a smaller custom q. Note that this is always safe: with all other parameters held fixed, decreasing q only increases the security level. This is very

[3] Bigger q means larger initial noise budget (good).
[4] Bigger q means lower security (bad).

easy in SEAL, as the user can access more than enough hard-coded primes q_i of various bit-length and of appropriate form through the functions `small_mods_60bit`, `small_mods_50bit`, `small_mods_40bit`, and `small_mods_30bit`.

The plaintext modulus t in SEAL is defined as a `SmallModulus` for performance reasons, and can therefore be any positive integer at least 2 and at most 60 bits in length. Note that when using batching (recall Section 7.4) t needs to be a prime such that $t = 1 \pmod{2n}$.

8.6 Encryption Parameter Qualifiers

After the encryption parameters are set, the instance of `EncryptionParameters` is given as input to the constructor of `SEALContext` to be evaluated for validity. In case the parameters are valid for homomorphic encryption, the instance of `SEALContext` is subsequently given to the constructors of tools such as `Encryptor` and `Decryptor`. Various properties of the parameters are stored in the `SEALContext` instance in a structure called `EncryptionParameterQualifiers`.

After the `SEALContext` is generated, the user can call `SEALContext::qualifiers` to return a copy of the qualifiers. Note that the only way to change the qualifiers is to change the encryption parameters themselves to support the particular features, and constructing a new `SEALContext`. In SEAL, `EncryptionParameterQualifiers` contains 5 qualifiers, which are described in Table 4.

Qualifier	Description
`parameters_set`	`true` if the encryption parameters are valid for SEAL, otherwise `false`.
`enable_fft`	`true` if n in polynomial modulus $x^n + 1$ is a power of 2, otherwise `false`.
`enable_ntt`	`true` if all NTT can be used for polynomial multiplication (see [26, 28]) with respect to all the factors q_i of q, otherwise `false`. See Section 8.5 for details.
`enable_batching`	`true` if batching (`PolyCRTBuilder`) can be used, otherwise `false`. See Section 7.4 for details.
`enable_fast_plain_lift`	`true` if all the small moduli $\{q_1, q_2, \ldots, q_k\}$ which construct the coefficient modulus are smaller than plaintext modulus t, otherwise `false`. If this is `true`, then `Evaluator::multiply_plain` becomes significantly faster.

Table 4: Encryption Parameter Qualifiers.

By far the most important of the qualifiers is `parameters_set`. In fact, if this is `true`, then `enable_fft` and `enable_ntt` must also be `true`. The qualifiers are mostly used internally to check whether the given parameters are compatible with specific operations and optimizations.

8.7 Automatic Parameter Selection

To assist the user in choosing parameters for a specific computation, SEAL 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 invariant noise $\|v\|$ (recall Section 6) 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 set to a heuristic upper bound estimate according to Table 2. `Simulator` is implemented in SEAL 2.3.1 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 7.2). `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 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 vector of `ChooserPoly` objects, a set `ParameterOptions` of pairs (n, q), a value for σ, and attempts to find an optimal pair $(n_{\text{opt}}, q_{\text{opt}})$ from `ParameterOptions`, together with an optimal value t_{opt}, such that that the parameters are just large enough to support the computations specified by all of the given `ChooserPoly` objects. It returns `true` if appropriate parameters were found, and populates a given instance of `EncryptionParameters` with $(x^{n_{\text{opt}}} + 1, q_{\text{opt}}, t_{\text{opt}})$. `SelectParameters` is implemented in SEAL 2.3.1 by the function `ChooserEvaluator::select_parameters`.

Recall from Section 8.3 that SEAL has an easy-to-access (and easy-to-modify) default set of pairs (n, q), and a default value for σ. The basic version of the function `ChooserEvaluator::select_parameters` uses these, but another overload lets custom values to be used instead. When calling `ChooserEvaluator::select_parameters`, both overloads require the user to give a *noise gap* g (in bits). The parameters are selected so that after the computations—with very high probability—there is at least g bits of noise budget left. To only ensure correctness, one can set the noise gap to 0.

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[5]. 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 eventually a good parameter set is found, `ChooserEvaluator::select_parameters` populates an instance of `EncryptionParameters` given to it, and returns `true`. Otherwise it returns `false`. An example demonstrating the automatic parameter selection tool is included with the library.

9 Security of FV

9.1 RLWE

The security of the FV encryption scheme is based on the apparent hardness of the famous *Ring Learning with Errors (RLWE)* problem [30]. We give a definition of the *decision*-RLWE problem appropriate to the rings that we use.

Definition 3 (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 $A_{s,\chi}$ 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 3. 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 [30, 19]), these proofs require the error terms e to be sampled from the distribution χ in a way very different from how SEAL does it. This, and performance reasons, is 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 [31]. Furthermore, it is possible to argue [25, 21] 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 2.3.1 (recall Section 5.9).

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 [31] and [21] for further details and discussion.

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

9.2 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 [32, 27]. 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. We are only aware of relatively small improvements to attacks of this type that utilize the ring structure in the RLWE samples.

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, 12, 1]) and the first draft of standardized (R)LWE parameter sets was proposed in [13]. The security estimates for the default parameters in Table 3 reflect best understanding at the time of writing [13], and should not be interpreted as definite security guarantees. We strongly recommend the user to consult experts in the security of (R)LWE when choosing parameters for SEAL, and in particular when using customized parameters.

9.3 Circular Security

Recall from Section 4 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 [21] this assumption is referred to as a form of *weak circular security*.

In SEAL 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.

9.4 Function Privacy

The privacy goal of SEAL is to allow the evaluation of arithmetic circuits on encrypted inputs, without revealing the input wire values to the evaluator. In particular, no attempt is made to keep any information hidden from the owner of the secret key. Even in a semi-honest security model this causes challenges for designing protocols (see e.g. [14]), since the evaluator might input some private information of its own to the circuit, which needs to be protected from the owner of the secret key. For example, 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, the highest power that was computed can be read from the size of the output ciphertext. A much bigger issue is that noise growth in homomorphic operations depends on the underlying plaintexts (recall Table 2): the owner of the secret key can compute the noise in the output ciphertext, and deduce information about the circuit, including the inputs of the evaluator.

It is possible to solve these problems and obtain *function privacy* [2] in a number of ways. One way already described by Gentry in [22] 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. An alternative approach, replacing flooding with a *soak-spin-repeat* strategy, is given by Ducas and Stehlé in [20]. This technique uses Gentry's bootstrapping process to repeatedly re-encrypt the ciphertext. Unfortunately this is slow, and requires the encryption parameters to be large enough to support bootstrapping (which is not currently implemented in SEAL). Finally, there are scheme specific function 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 [24] is described in [6].

Due to its superior performance, we recommend using the noise flooding technique when necessary. In practice, a "smudging lemma" (see e.g. [3]) can be used together with the heuristic noise growth estimates implemented in SEAL to precisely bound the amount of noise that needs to be flooded to obtain a given statistical security level. For a concrete example, we refer the reader to [14].

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] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 483–501. Springer, 2012.

[4] 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`.

[5] 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.

[6] Florian Bourse, Rafaël Del Pino, Michele Minelli, and Hoeteck Wee. FHE circuit privacy almost for free. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 62–89. Springer, 2016.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, July 2017.

[14] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption, 2017.

[15] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Technical report, IACR Cryptology ePrint Archive, 2016: 421, 2016.
[16] Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In 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*, pages 325–340. Springer, 2016.
[17] Eric Crockett and Chris Peikert. Challenges for ring-LWE. Cryptology ePrint Archive, Report 2016/782, 2016. `http://eprint.iacr.org/2016/782`.
[18] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. *Proceedings of the IEEE*, 105(3), 2017.
[19] 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.
[20] 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.
[21] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `http://eprint.iacr.org/2012/144`.
[22] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.
[23] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.
[24] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.
[25] Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Robustness of the learning with errors assumption. 2010.
[26] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014.
[27] 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.
[28] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.
[29] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.
[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] 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.
[32] 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.
[33] Nigel P Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

Appendix

Initial Noise

Lemma 9. *Let $\mathtt{ct} = (c_0, c_1)$ be a fresh encryption of a message $m \in R_t$. The noise v in $\mathtt{ct}$ satisfies*

$$\|v\| \leq \frac{r_t(q)}{q}\|m\| + \frac{tB}{q}(2n+1)\,.$$

Proof. Let $\texttt{ct} = (c_0, c_1)$ be an encryption of m under the public key $\texttt{pk} = (p_0, p_1) = ([-(as + e)]_q, a)$. Then, for some polynomials k_0, k_1, k,

$$\begin{aligned}
\frac{t}{q}(c_0 + c_1 s) &= \frac{t}{q}(\Delta m + p_0 u + e_0 + k_0 q + p_1 us + e_1 s + k_1 qs) \\
&= m + \frac{t}{q}\left(\frac{-r_t(q)m}{t} + p_0 u + e_0 + p_1 us + e_1 s\right) + t(k_0 + k_1 s) \\
&= m + \frac{t}{q}\left(\frac{-r_t(q)}{t}m + (-as - e + kq)u + e_0 + aus + e_1 s\right) + t(k_0 + k_1 s) \\
&= m + \frac{t}{q}\left(\frac{-r_t(q)}{t}m - eu + e_1 + e_2 s\right) + t(k_0 + k_1 s + ku)\,,
\end{aligned}$$

so the noise is

$$v = \frac{t}{q}\left(\frac{-r_t(q)}{t}m - eu + e_1 + e_2 s\right).$$

To bound $\|v\|$, we use the fact that the error polynomials sampled from χ have coefficients bounded by B, and that $\|s\| = \|u\| = 1$. Then

$$\|v\| \leq \frac{r_t(q)}{q}\|m\| + \frac{tB}{q}(2n + 1)\,.$$

□

Addition

Lemma 10. *Let $\texttt{ct}_1 = (c_0, c_1, \ldots, c_j)$ and $\texttt{ct}_2 = (d_0, d_1, \ldots, d_k)$ be two ciphertexts encrypting $m_1, m_2 \in R_t$, and having noises v_1, v_2, respectively. Then the noise v_{add} in their sum $\texttt{ct}_{add}$ is $v_{add} = v_1 + v_2$, and satisfies $\|v_{add}\| \leq \|v_1\| + \|v_2\|$.*

Proof. By definition of homomorphic addition, $\texttt{ct}_{\text{add}}$ encrypts $[m_1 + m_2]_t$. Let $[m_1 + m_2]_t = m_1 + m_2 + at$ for some integer coefficient polynomial a. Suppose WLOG that $\max(j, k) = j$, so that

$$\texttt{ct}_{\text{add}} = (c_0 + d_0, \ldots, c_k + d_k, c_{k+1}, \ldots c_j)\,.$$

By definition of noise in $\texttt{ct}_1$ and $\texttt{ct}_2$, we have

$$\frac{t}{q}\texttt{ct}_1(s) = m_1 + v_1 + a_1 t\,, \quad \frac{t}{q}\texttt{ct}_2(s) = m_2 + v_2 + a_2 t\,,$$

for some polynomials a_1, a_2 with integer coefficients. Therefore

$$\begin{aligned}
\frac{t}{q}\texttt{ct}_{\text{add}}(s) &= \frac{t}{q}\texttt{ct}_1(s) + \frac{t}{q}\texttt{ct}_2(s) \\
&= m_1 + v_1 + a_1 t + m_2 + v_2 + a_2 t \\
&= [m_1 + m_2]_t + (m_1 + m_2 - [m_1 + m_2]_t) + v_1 + v_2 + (a_1 + a_2)t \\
&= [m_1 + m_2]_t + v_1 + v_2 + (a_1 + a_2 - a)t\,,
\end{aligned}$$

so the noise is $v_{\text{add}} = v_1 + v_2$, and $\|v_{\text{add}}\| = \|v_1 + v_2\| \leq \|v_1\| + \|v_2\|$. □

Multiplication

Lemma 11. *Let $\mathtt{ct}_1 = (x_0, \dots, x_{j_1})$ be a ciphertext of size j_1+1 encrypting m_1 with noise v_1, and let $\mathtt{ct}_2 = (y_0, \dots, y_{j_2})$ be a ciphertext of size j_2+1 encrypting m_2 with noise v_2. Let N_{m_1} and N_{m_2} be upper bounds on the number of non-zero terms in the polynomials m_1 and m_2, respectively. Then the noise v_{mult} in the product $\mathtt{ct}_{mult}$ satisfies the following bound:*

$$\begin{aligned}\|v_{mult}\| \leq &\left[(N_{m_1}+n)\|m_1\| + \frac{nt}{2}\cdot\frac{n^{j_1+1}-1}{n-1}\right]\|v_2\| \\ &+ \left[(N_{m_2}+n)\|m_2\| + \frac{nt}{2}\cdot\frac{n^{j_2+1}-1}{n-1}\right]\|v_1\| \\ &+ 3n\|v_1\|\|v_2\| + \frac{t}{2q}\left(\frac{n^{j_1+j_2+1}-1}{n-1}\right).\end{aligned}$$

Proof. By definition of homomorphic multiplication the ciphertext $\mathtt{ct}_{\text{mult}} = (c_0, \dots c_{j_1+j_2})$ is such that for $0 \leq i \leq j_1+j_2$, for some polynomials ϵ_i with coefficients in $(-\frac{1}{2}, \frac{1}{2}]$, and for some polynomials A_i with integer coefficients,

$$c_i = \left[\left\lfloor \frac{t}{q}\left(\sum_{k+l=i} x_k y_l\right)\right\rceil\right]_q = \left\lfloor \frac{t}{q}\left(\sum_{k+l=i} x_k y_l\right)\right\rceil + A_i q = \frac{t}{q}\left(\sum_{k+l=i} x_k y_l\right) + \epsilon_i + A_i q\,.$$

Also, by definition $\mathtt{ct}_{\text{mult}}$ encrypts $[m_1m_2]_t$, and that $[m_1m_2]_t = m_1m_2 + at$ for some polynomial a with integer coefficients.

By definition of noise in $\mathtt{ct}_1$ and $\mathtt{ct}_2$, we have for some polynomials a_1, a_2 with integer coefficients,

$$\frac{t}{q}\mathtt{ct}_1(s) = m_1 + v_1 + a_1 t\,, \quad \frac{t}{q}\mathtt{ct}_2(s) = m_2 + v_2 + a_2 t\,.$$

We then compute

$$\begin{aligned}\frac{t}{q}\mathtt{ct}_{\text{mult}}(s) &= \frac{t}{q}(c_0, \dots, c_{j_1+j_2})(s) \\ &= \frac{t}{q}\left[\left(\frac{t}{q}(x_0y_0) + \epsilon_0 + A_0 q\right) + \dots + \left(\frac{t}{q}(x_{j_1}y_{j_2}) + \epsilon_{j_1+j_2} + A_{j_1+j_2}q\right) s^{j_1+j_2}\right] \\ &= \frac{t}{q}\cdot\frac{t}{q}\left[\sum_{i=0}^{j_1+j_2}\left(\sum_{k+l=i} x_k y_l\right)s^i\right] + \frac{t}{q}\sum_{i=0}^{j_1+j_2}\epsilon_i s^i + \left(\sum_{i=0}^{j_1+j_2} A_i s^i\right)t \\ &= \frac{t}{q}\mathtt{ct}_1(s)\cdot\frac{t}{q}\mathtt{ct}_2(s) + \frac{t}{q}\sum_{i=0}^{j_1+j_2}\epsilon_i s^i + \left(\sum_{i=0}^{j_1+j_2} A_i s^i\right)t \\ &= (m_1 + v_1 + a_1 t)(m_2 + v_2 + a_2 t) + \frac{t}{q}\sum_{i=0}^{j_1+j_2}\epsilon_i s^i + \left(\sum_{i=0}^{j_1+j_2} A_i s^i\right)t \\ &= [m_1m_2]_t + m_1v_2 + m_2v_1 + v_1v_2 + v_1a_2t + v_2a_1t + \frac{t}{q}\sum_{i=0}^{j_1+j_2}\epsilon_i s^i \\ &\quad + \left(m_1a_2 + m_2a_1 + a_1a_2t + \sum_{i=0}^{j_1+j_2} A_i s^i - a\right)t\,,\end{aligned}$$

where in the last step we used $m_1 m_2 = [m_1 m_2]_t - at$. Thus, we find that the noise in $\mathtt{ct}_{\mathrm{mult}}$ is given by

$$v_{\mathrm{mult}} = m_1 v_2 + m_2 v_1 + v_1 v_2 + (v_1 a_2 + v_2 a_1)t + \frac{t}{q} \sum_{i=0}^{j_1+j_2} \epsilon_i s^i \,.$$

To be able to bound the new noise, we first note that

$$\frac{t}{q} \left\| \sum_{i=0}^{j_1+j_2} \epsilon_i s^i \right\| \leq \frac{t}{2q} \left(\frac{n^{j_1+j_2+1} - 1}{n-1} \right) . \tag{1}$$

Next, we write $a_i t = \frac{t}{q} \mathtt{ct}_i(s) - m_i - v_i$, and note that

$$\|a_i t\| \leq \frac{t}{2} \cdot \frac{n^{j_i+1} - 1}{n-1} + \|m_i\| + \|v_i\| \,. \tag{2}$$

Finally, using (1) and (2) we can bound the noise growth in multiplication:

$$\begin{aligned}
\|v_{\mathrm{mult}}\| &= \left\| m_1 v_2 + m_2 v_1 + v_1 v_2 + (v_1 a_2 + v_2 a_1)t + \frac{t}{q} \sum_{i=0}^{j_1+j_2} \epsilon_i s^i \right\| \\
&\leq \|m_1 v_2\| + \|m_2 v_1\| + \|v_1 v_2\| + \|(v_1 a_2 + v_2 a_1)t\| + \frac{t}{q} \left\| \sum_{i=0}^{j_1+j_2} \epsilon_i s^i \right\| \\
&\leq N_{m_1} \|m_1\| \|v_2\| + N_{m_2} \|m_2\| \|v_1\| + n \|v_1\| \|v_2\| \\
&\quad + n\|v_1\| \left(\frac{t}{2} \cdot \frac{n^{j_2+1} - 1}{n-1} + \|m_2\| + \|v_2\| \right) \\
&\quad + n\|v_2\| \left(\frac{t}{2} \cdot \frac{n^{j_1+1} - 1}{n-1} + \|m_1\| + \|v_1\| \right) + \frac{t}{2q} \left(\frac{n^{j_1+j_2+1} - 1}{n-1} \right) \\
&= \left[(N_{m_1} + n)\|m_1\| + \frac{nt}{2} \cdot \frac{n^{j_1+1} - 1}{n-1} \right] \|v_2\| \\
&\quad + \left[(N_{m_2} + n)\|m_2\| + \frac{nt}{2} \cdot \frac{n^{j_2+1} - 1}{n-1} \right] \|v_1\| \\
&\quad + 3n\|v_1\|\|v_2\| + \frac{t}{2q} \left(\frac{n^{j_1+j_2+1} - 1}{n-1} \right) .
\end{aligned}$$

□

Relinearization

Lemma 12. *Let* $\mathtt{ct}$ *be a ciphertext of size* $M+1$ *encrypting* m*, and having noise* v*. Let* $\mathtt{ct}_{relin}$ *of size* $N+1$ *be the ciphertext encrypting* m*, obtained by the relinearization of* $\mathtt{ct}$*, where* $2 \leq N+1 < M+1$*. Then, the noise* v_{relin} *in* $\mathtt{ct}_{relin}$ *is given by*

$$v_{relin} = v - \frac{t}{q} \sum_{j=0}^{M-N-1} \sum_{i=0}^{\ell} e_{(M-j),i} c_{M-j}^{(i)} \,,$$

and can be bounded as

$$\|v_{relin}\| \leq \|v\| + \frac{t}{q}(M-N)nB(\ell+1)w \,.$$

Proof. Relinearization of a ciphertext from size $M+1$ to size $N+1$, where $2 \le N+1 < M+1$ consists of $M-N$ 'one-step' relinearizations. In each step, the 'current' ciphertext $(c_0, c_1, \ldots, c_k)$ is transformed to an intermediate ciphertext $\texttt{ct}' = (c_0', c_1', \ldots, c_{k-1}')$ using the appropriate evaluation key

$$\texttt{evk}_k = [([-(a_{k,i}s + e_{k,i}) + w^i s^k]_q, a_{k,i}) : i = 0, \ldots, \ell]\,.$$

In the following step, $\texttt{ct}'$ becomes the 'current ciphertext', and so on until the intermediate ciphertext produced is of size $N+1$, at which point it is output as $\texttt{ct}_{\text{relin}}$.

The input ciphertext is $\texttt{ct} = (c_0, c_1, \ldots, c_M)$, and after the first one-step relinearization, the intermediate ciphertext is $\texttt{ct}' = (c_0', c_1', \ldots, c_{M-1}')$, where

$$c_0' = c_0 + \sum_{i=0}^{\ell} \texttt{evk}_M[i][0] c_M^{(i)}\,, \quad c_1' = c_1 + \sum_{i=0}^{\ell} \texttt{evk}_M[i][1] c_M^{(i)}\,,$$

and $c_j' = c_j$ for $2 \le j \le M-1$. So, for some polynomials a_i with integer coefficients, where $0 \le i \le \ell+1$,

$$\begin{aligned}
\frac{t}{q}\texttt{ct}'(s) &= \frac{t}{q}\left(c_0' + c_1's + \ldots + c_{M-1}'s^{M-1}\right) \\
&= \frac{t}{q}\left[c_0 + \sum_{i=0}^{\ell} \texttt{evk}_M[i][0]c_M^{(i)} + \left(c_1 + \sum_{i=0}^{\ell} \texttt{evk}_M[i][1]c_M^{(i)}\right)s + \ldots + c_{M-1}s^{M-1}\right] \\
&= \frac{t}{q}\left(\sum_{i=0}^{\ell} \texttt{evk}_M[i][0]c_M^{(i)} + s\sum_{i=0}^{\ell} \texttt{evk}_M[i][1]c_M^{(i)}\right) + \frac{t}{q}\left(c_0 + c_1 s + \ldots + c_{M-1}s^{M-1}\right) \\
&= \frac{t}{q}\left(-\sum_{i=0}^{\ell} e_{M,i}c_M^{(i)} + \sum_{i=0}^{\ell} a_i q c_M^{(i)} + s^M \sum_{i=0}^{\ell} w^i c_M^{(i)}\right) + \frac{t}{q}\left(c_0 + c_1 s + \ldots + c_{M-1}s^{M-1}\right) \\
&= \frac{t}{q}\left(-\sum_{i=0}^{\ell} e_{M,i}c_M^{(i)} + \sum_{i=0}^{\ell} a_i q c_M^{(i)}\right) + \frac{t}{q}s^M c_M + \frac{t}{q}\left(c_0 + c_1 s + \ldots + c_{M-1}s^{M-1}\right) \\
&= -\frac{t}{q}\sum_{i=0}^{\ell} e_{M,i}c_M^{(i)} + \frac{t}{q}\left(c_0 + c_1 s + \ldots + c_{M-1}s^{M-1} + c_M s^M\right) + t\sum_{i=0}^{\ell} a_i c_M^{(i)} \\
&= m + v - \frac{t}{q}\sum_{i=0}^{\ell} e_{M,i}c_M^{(i)} + \left(a_{\ell+1} + \sum_{i=0}^{\ell} a_i c_M^{(i)}\right)t\,.
\end{aligned}$$

Hence, the noise grows by an additive factor $-\frac{t}{q}\sum_{i=0}^{\ell} e_{M,i}c_M^{(i)}$ in a one-step relinearization. Iterating this process, we find the noise after relinearization:

$$v_{\text{relin}} = v - \frac{t}{q}\sum_{j=0}^{M-N-1}\sum_{i=0}^{\ell} e_{M-j,i}c_{M-j}^{(i)}\,.$$

Bounding $\|v_{\text{relin}}\|$ is easy:

$$\begin{aligned}\|v_{\text{relin}}\| &= \left\| v - \frac{t}{q} \sum_{j=0}^{M-N-1} \sum_{i=0}^{\ell} e_{M-j,i} c_{M-j}^{(i)} \right\| \\ &\leq \|v\| + \frac{t}{q} \sum_{j=0}^{M-N-1} \sum_{i=0}^{\ell} \left\| e_{M-j,i} c_{M-j}^{(i)} \right\| \\ &\leq \|v\| + \frac{t}{q}(M-N)nB(\ell+1)w\,.\end{aligned}$$

□

Plain Multiplication

Lemma 13. *Let* $\mathtt{ct} = (x_0, \ldots, x_j)$ *be a ciphertext encrypting* m_1 *with noise* v*, and let* m_2 *be a plaintext polynomial. Let* N_{m_2} *be an upper bound on the number of non-zero terms in the polynomial* m_2*. Let* $\mathtt{ct}_{pmult}$ *denote the ciphertext obtained by plain multiplication of* $\mathtt{ct}$ *with* m_2*. Then the noise in the plain product* $\mathtt{ct}_{pmult}$ *is* $v_{pmult} = m_2 v$*, and we have the bound*

$$\|v_{pmult}\| \leq N_{m_2} \|m_2\| \|v\|\,.$$

Proof. By definition the ciphertext $\mathtt{ct}_{\text{pmult}} = (m_2 x_0, \ldots, m_2 x_j)$. Hence for some polynomials a, a' with integer coefficients,

$$\begin{aligned}\frac{t}{q}\mathtt{ct}_{\text{pmult}}(s) &= \frac{t}{q}\left(m_2 x_0 + m_2 x_1 s + \cdots + m_2 x_j s^j\right) \\ &= m_2 \frac{t}{q}\left(x_0 + x_1 s + \cdots + x_j s^j\right) \\ &= m_2 \frac{t}{q}\mathtt{ct}(s) \\ &= m_2(m_1 + v + at) \\ &= m_1 m_2 + m_2 v + m_2 at \\ &= [m_1 m_2]_t + m_2 v + (m_2 a - a')t\,,\end{aligned}$$

where in the last line we used $[m_1 m_2]_t = m_1 m_2 + a' t$. Hence the noise is $v_{\text{pmult}} = m_2 v$ and can be bounded as

$$\|v_{\text{pmult}}\| \leq N_{m_2} \|m_2\| \|v\|\,.$$

□

Plain Addition

Lemma 14. *Let* $\mathtt{ct} = (x_0, \ldots, x_j)$ *be a ciphertext encrypting* m_1 *with noise* v*, and let* m_2 *be a plaintext polynomial. Let* $\mathtt{ct}_{padd}$ *denote the ciphertext obtained by plain addition of* $\mathtt{ct}$ *with* m_2*. Then the noise in* $\mathtt{ct}_{padd}$ *is* $v_{padd} = v - \frac{r_t(q)}{q} m_2$*, and we have the bound*

$$\|v_{padd}\| \leq \|v\| + \frac{r_t(q)}{q} \|m_2\|\,.$$

Proof. By definition of plain addition we have $\mathtt{ct}_{\text{padd}} = (x_0 + \Delta m_2, x_1, \ldots, x_j)$. Hence for some polynomials a, a' with integer coefficients,

$$\begin{aligned}
\frac{t}{q}\mathtt{ct}_{\text{padd}}(s) &= \frac{t}{q}\left(x_0 + \Delta m_2 + x_1 s + \cdots + x_j s^j\right) \\
&= \frac{\Delta t}{q} m_2 + \frac{t}{q}\left(x_0 + x_1 s + \cdots + x_j s^j\right) \\
&= \frac{\Delta t}{q} m_2 + \frac{t}{q}\mathtt{ct}(s) \\
&= m_1 + v + \frac{q - r_t(q)}{q} m_2 + at \\
&= m_1 + m_2 + v - \frac{r_t(q)}{q} m_2 + at \\
&= [m_1 + m_2]_t + v - \frac{r_t(q)}{q} m_2 + (a - a')t\,,
\end{aligned}$$

where in the last line we used $[m_1 + m_2]_t = m_1 + m_2 + a't$. Hence the noise is

$$v_{\text{padd}} = v - \frac{r_t(q)}{q} m_2$$

and this can be bounded as

$$\|v_{\text{padd}}\| \leq \|v\| + \frac{r_t(q)}{q}\|m_2\|\,.$$

□

Negation

Lemma 15. *Let* $\mathtt{ct}$ *be a ciphertext encrypting* m *with noise* v *and* $\mathtt{ct}_{neg}$ *be its negation. The noise* v_{neg} *in* $\mathtt{ct}_{neg}$ *is given by* $v_{neg} = -v$ *and we have*

$$\|v_{neg}\| = \|v\|\,.$$

Proof. If $\mathtt{ct} = (c_0, c_1, \ldots, c_k)$ then its negation $\mathtt{ct}_{\text{neg}} = (-c_0, -c_1, \ldots, -c_k) = -(c_0, c_1, \ldots, c_k)$. So

$$\begin{aligned}
\frac{t}{q}\mathtt{ct}_{\text{neg}}(s) &= -\frac{t}{q}\mathtt{ct}(s) \\
&= -(m + v + at) \\
&= -m + (-v) + (-a)t.
\end{aligned}$$

Hence the noise v_{neg} in $\mathtt{ct}_{\text{neg}}$ is $-v$ and $\|v_{\text{neg}}\| = \|v\|$. □

Subtraction

Suppose $\mathtt{ct}_1$ and $\mathtt{ct}_2$ are two ciphertexts encrypting m_1 and m_2 and we want to compute a ciphertext $\mathtt{ct}_{sub}$ encrypting $m_1 - m_2$. We could firstly negate $\mathtt{ct}_2$ to obtain a ciphertext $\mathtt{ct}_2'$ that encrypts $-m_2$ and then perform an addition of $\mathtt{ct}_1$ and $\mathtt{ct}_2'$. By viewing the subtraction operation in this way we can see that the noise growth in subtraction is at most that for addition, since the noise does not change in norm in negation.

Lemma 16. *Let* $\mathtt{ct}_1$ *and* $\mathtt{ct}_2$ *be two ciphertexts encrypting* m_1, m_2 *respectively with noises* v_1,v_2 *respectively. The noise* v_{sub} *in the result* $\mathtt{ct}_{sub}$ *is bounded as* $\|v_{sub}\| \leq \|v_1\| + \|v_2\|$.

Plain subtraction

By the same argument as for subtraction, the noise growth in plain subtraction is at most that for plain addition.

Lemma 17. *Let* $\mathtt{ct}$ *be a ciphertext encrypting* m_1 *with noise* v*, and let* m_2 *be a plaintext polynomial. Let* $\mathtt{ct}_{psub}$ *denote the ciphertext obtained by plain subtraction of* m_2 *from* $\mathtt{ct}$*. Then the noise* v_{psub} *in* $\mathtt{ct}_{psub}$ *is bounded as*

$$\|v_{psub}\| \leq \|v\| + \frac{r_t(q)}{q} \|m_2\| \, .$$