Design and Implementation of a Homomorphic-Encryption Library

Shai Halevi Victor Shoup

April 11, 2013

Abstract

We describe the design and implementation of a software library that implements the Brakerski-Gentry-Vaikuntanathan (BGV) homomorphic encryption scheme, along with many optimizations to make homomorphic evaluation runs faster, focusing mostly on effective use of the Smart-Vercauteren ciphertext packing techniques. Our library is written in C++ and uses the NTL mathematical library. It is distributed under the terms of the GNU General Public License (GPL).

Partially supported by DARPA under agreement number FA8750-11-C-0096. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

Also partially supported by the Intelligence Advanced Research Projects Activity (EARP) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

Contents

Organization of This Report

We begin in Section 1 with a brief high-level overview of the BGV cryptosystem and some important features of the variant that we implemented and our choice of representation, as well as an overview of the structure of our library. Then in Sections 2, 3,4 we give a bottom-up detailed description of all the modules in the library. We conclude in Section 5 with some examples of using this library.

1 The BGV Homomorphic Encryption Scheme

A homomorphic encryption scheme [8, 3] allows processing of encrypted data even without knowing the secret decryption key. In this report we describe the design and implementation of a software library that implements the Brakerski-Gentry-Vaikuntanathan (BGV) homomorphic encryption scheme [2]. We begin by a high-level description of the the BGV variant that we implemented, followed by a detailed description of the various software components in our implementation. The description in this section is mostly taken from the full version of [5].

Below we denote by $[\cdot]_q$ the reduction-mod-q function, namely mapping an integer $z \in \mathbb{Z}$ to the unique representative of its equivalence class modulo q in the interval $(-q/2, q/2]$. We use the same notation for modular reduction of vectors, matrices, etc.

Our BGV variant is defined over polynomial rings of the form $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$ where m is a parameter and $\Phi_m(X)$ is the m'th cyclotomic polynomial. The "native" plaintext space for this scheme is usually the ring $\mathbb{A}_2 = \mathbb{A}/2\mathbb{A}$, namely binary polynomials modulo $\Phi_m(X)$. (Our implementation supports other plaintext spaces as well, but in this report we mainly describe the case of plaintext space $\mathbb{A}_2$. See some more details in Section 2.4.) We use the Smart-Vercauteren CRT-based encoding technique [10] to "pack" a vector of bits in a binary polynomial, so that polynomial arithmetic in $\mathbb{A}_2$ translates to entry-wise arithmetic on the packed bits.

The ciphertext space for this scheme consists of vectors over $\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$, where q is an odd modulus that evolves with the homomorphic evaluation. Specifically, the system is parametrized by a "chain" of moduli of decreasing size, $q_0 < q_1 < \cdots < q_L$ and freshly encrypted ciphertexts are defined over R_{q_L}. During homomorphic evaluation we keep switching to smaller and smaller moduli until we get ciphertexts over $\mathbb{A}_{q_0}$, on which we cannot compute anymore. We call ciphertexts that are defined over $\mathbb{A}_{q_i}$ "level-i ciphertexts". These level-i ciphertexts are 2-element vectors over R_{q_i}, i.e., $\vec{c} = (c_0, c_1) \in (\mathbb{A}_{q_i})^2$.

Secret keys are polynomials $\mathfrak{s} \in \mathbb{A}$ with "small" coefficients, and we view $\mathfrak{s}$ as the second element of the 2-vector $\vec{s} = (1, \mathfrak{s})$. A level-$i$ ciphertext $\vec{c} = (c_0, c_1)$ encrypts a plaintext polynomial $m \in \mathbb{A}_2$ with respect to $\vec{s} = (1, \mathfrak{s})$ if we have the equality over $\mathbb{A}$, $[\langle \vec{c}, \vec{s} \rangle]_{q_i} = [c_0 + \mathfrak{s} \cdot c_1]_{q_i} \equiv m \pmod 2$, and moreover the polynomial $[c_0 + \mathfrak{s} \cdot c_1]_{q_i}$ is "small", i.e. all its coefficients are considerably smaller than q_i. Roughly, that polynomial is considered the "noise" in the ciphertext, and its coefficients grow as homomorphic operations are performed. We note that the crux of the noise-control technique from [2] is that a level-i ciphertext can be publicly converted into a level-$(i+1)$ ciphertext (with respect to the same secret key), and that this transformation reduces the noise in the ciphertext roughly by a factor of q_{i+1}/q_i.

Following [7, 4, 5], we think of the "size" of a polynomial $a \in \mathbb{A}$ as the norm of its canonical embedding. Recall that the canonical embedding of $a \in \mathbb{A}$ into $\mathbb{C}^{\phi(m)}$ is the $\phi(m)$-vector of complex numbers $\sigma(a) = (a(\tau_m^j))_j$ where τ_m is a complex primitive m-th root of unity ($\tau_m = e^{2\pi i/m}$) and the indexes j range over all of $\mathbb{Z}_m^*$. We denote the l_2-norm of the canonical embedding of a by

$\|a\|_2^{\text{canon}}$.

The basic operations that we have in this scheme are the usual key-generation, encryption, and decryption, the homomorphic evaluation routines for addition, multiplication and automorphism (and also addition-of-constant and multiplication-by-constant), and the "ciphertext maintenance" operations of key-switching and modulus-switching. These are described in the rest of this report, but first we describe our plaintext encoding conventions and our Double-CRT representation of polynomials.

1.1 Plaintext Slots

The native plaintext space of our variant of BGV are elements of $\mathbb{A}_2$, and the polynomial $\Phi_m(X)$ factors modulo 2 into ℓ irreducible factors, $\Phi_m(X) = F_1(X) \cdot F_2(X) \cdots F_\ell(X) \pmod 2$, all of degree $d = \phi(m)/\ell$. Just as in [2, 4, 10] each factor corresponds to a "plaintext slot". That is, we can view a polynomial $a \in \mathbb{A}_2$ as representing an ℓ-vector $(a \bmod F_i)_{i=1}^{\ell}$.

More specifically, for the purpose of packing we think of a polynomial $a \in \mathbb{A}_2$ not as a binary polynomial but as a polynomial over the extension field $\mathbb{F}_{2^d}$ (with some specific representation of that field), and the plaintext values that are encoded in a are its evaluations at ℓ specific primitive m-th roots of unity in $\mathbb{F}_{2^d}$. In other words, if $\rho \in \mathbb{F}_{2^d}$ is a particular fixed primitive m-th root of unity, and our distinguished evaluation points are $\rho^{t_1}, \rho^{t_2}, \ldots, \rho^{t_\ell}$ (for some set of indexes $T = \{t_1, \ldots, t_\ell\}$), then the vector of plaintext values encoded in a is:

$$\big(a(\rho^{t_j}) \;:\; t_j \in T\big).$$

See Section 2.4 for a discussion of the choice of representation of $\mathbb{F}_{2^d}$ and the evaluation points.

It is a standard fact that (for an abstract primitive m-th root of unity ζ_m), the Galois group $\mathcal{G}\mathsf{al} = \mathcal{G}\mathsf{al}(\mathbb{Q}(\zeta_m)/\mathbb{Q})$ consists of the mappings $\kappa_k : a(X) \mapsto a(X^k) \bmod \Phi_m(X)$ for all k co-prime with m, and that it is isomorphic to $\mathbb{Z}_m^*$. As noted in [4], for each $i, j \in \{1, 2, \ldots, \ell\}$ there is an element $\kappa_k \in \mathcal{G}\mathsf{al}$ which sends an element in slot i to an element in slot j. Indeed if we set $k = t_j^{-1} \cdot t_i \pmod m$ and $b = \kappa_k(a)$ then we have

$$b(\rho^{t_j}) = a(\rho^{t_j k}) = a(\rho^{t_j \cdot t_j^{-1} t_i}) = a(\rho^{t_i}),$$

so the element in the j'th slot of b is the same as that in the i'th slot of a. In addition to these "data-movement maps", $\mathcal{G}\mathsf{al}$ contains also the Frobenius maps, $X \longrightarrow X^{2^i}$, which also act as Frobenius on the individual slots.

We note that the values that are encoded in the slots do not have to be individual bits, in general they can be elements of the extension field $\mathbb{F}_{2^d}$ (or any sub-field of it). For example, for the AES application we may want to pack elements of $\mathbb{F}_{2^8}$ in the slots, so we choose the parameters so that $\mathbb{F}_{2^8}$ is a sub-field of $\mathbb{F}_{2^d}$ (which means that d is divisible by 8).

More generally, we allow plaintext spaces of the form $\mathbb{A}_{p^r}$, rather than just $\mathbb{A}_2$, where p is an arbitrary prime (which does not divide m) and r is a small positive integer. In this setting, $\Phi_m(X)$ factors as $\prod_{i=1}^{\ell} F_i$ modulo $\mathbb{Z}_{p^r}$, where each F_i is irreducible modulo p^r of the same degree d. Note that in the case $r > 1$, the factorization of $\Phi_m(X)$ is determined by Hensel lifting. The ith plaintext slot is isomorphic as a $\mathbb{Z}_{p^r}$-algebra to $\mathbb{Z}_{p^r}/(F_i)$. It turns out that all these different rings are isomorphic — in the case where $r = 1$, they are isomorphic to the finite field $\mathbb{F}_{p^d}$.

1.2 Our Modulus Chain and Double-CRT Representation

We define the chain of moduli by choosing $L+1$ "small primes" $p_0, p_1, \ldots, p_L$ and the l'th modulus in our chain is defined as $q_l = \prod_{j=0}^{l} p_j$. The primes p_i's are chosen so that for all i, $\mathbb{Z}/p_i\mathbb{Z}$ contains a primitive m-th root of unity (call it ζ_i) so $\Phi_m(X)$ factors modulo p_i to linear terms $\Phi_m(X) = \prod_{j \in \mathbb{Z}_m^*}(X - \zeta_i^j) \pmod{p_i}$.

A key feature of our implementation is that we represent an element $a \in \mathbb{A}_{q_l}$ via double-CRT representation, with respect to both the integer factors of q_l and the polynomial factors of $\Phi_m(X)$ mod q_l. A polynomial $a \in \mathbb{A}_q$ is represented as the $(l+1) \times \phi(m)$ matrix of its evaluation at the roots of $\Phi_m(X)$ modulo p_i for $i = 0, \ldots, l$:

$$\mathsf{DoubleCRT}^l(a) = \Big(a(\zeta_i^j) \bmod p_i\Big)_{0 \le i \le l,\ j \in \mathbb{Z}_m^*}.$$

Addition and multiplication in $\mathbb{A}_q$ can be computed as component-wise addition and multiplication of the entries in the two tables (modulo the appropriate primes p_i),

$$\begin{aligned}\mathsf{DoubleCRT}^l(a+b) &= \mathsf{DoubleCRT}^l(a) + \mathsf{DoubleCRT}^l(b),\\ \mathsf{DoubleCRT}^l(a \cdot b) &= \mathsf{DoubleCRT}^l(a) \cdot \mathsf{DoubleCRT}^l(b).\end{aligned}$$

Also, for an element of the Galois group $\kappa \in \mathcal{G}\mathsf{al}$, mapping $a(X) \in \mathbb{A}$ to $a(X^k) \bmod \Phi_m(X)$, we can evaluate $\kappa(a)$ on the double-CRT representation of a just by permuting the columns in the matrix, sending each column j to column $j \cdot k \bmod m$.

1.3 Modules in our Library

Very roughly, our HE library consists of four layers: in the bottom layer we have modules for implementing mathematical structures and various other utilities, the second layer implements our Double-CRT representation of polynomials, the third layer implements the cryptosystem itself (with the "native" plaintext space of binary polynomials), and the top layer provides interfaces for using the cryptosystem to operate on arrays of plaintext values (using the plaintext slots as described in Section 1.1). We think of the bottom two layers as the "math layers", and the top two layers as the "crypto layers", and describe then in detail in Sections 2 and 3, respectively. A block-diagram description of the library is given in Figure 1. Roughly, the modules NumbTh, `timing`, bluestein, PAlgebra, PAlgebraMod, Cmodulus, IndexSet and IndexMap belong to the bottom layer, FHEcontext, SingleCRT and DoubleCRT belong to the second layer, FHE, Ctxt and KeySwitching are in the third layer, and EncryptedArray is in the top layer.

2 The Math Layers

2.1 The `timing` module

This module contains some utility functions for measuring the time that various methods take to execute. To use it, we insert the macro `FHE_TIMER_START` at the beginning of the method(s) that we want to time and `FHE_TIMER_STOP` at the end, then the main program needs to call the function `setTimersOn()` to activate the timers and `setTimersOff()` to pause them. We can have at most one timer per method/function, and the timer is called by the same name as the function itself (using the built-in macro `__func__`). To obtain the value of a given timer (in seconds), the

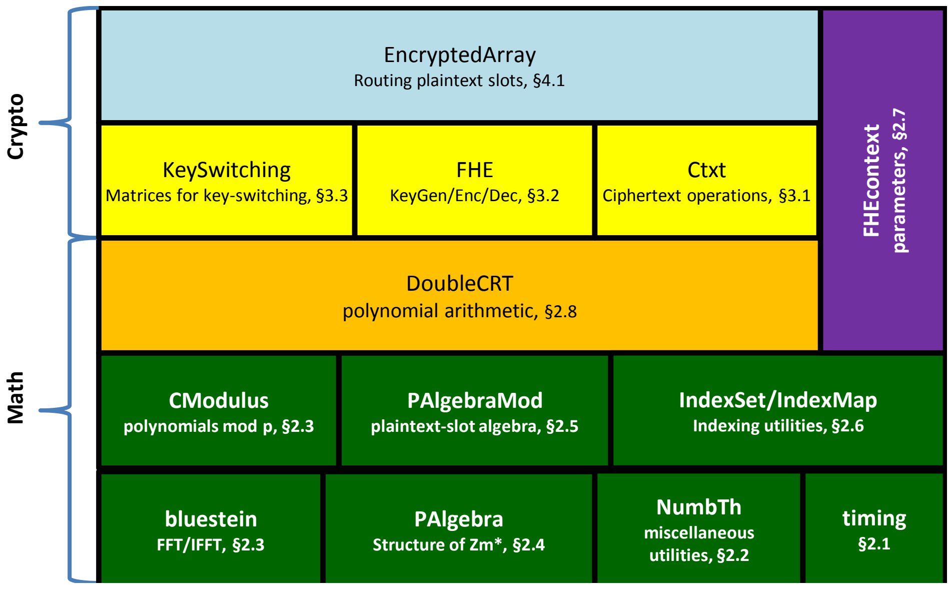

Figure 1: A block diagram of the Homomorphic-Encryption library

application can use the function `double getTime4func(const char *fncName)`, and the function `printAllTimers()` prints the values of all timers to the standard output.

2.2 NumbTh: Miscellaneous Utilities

This module started out as an implementation of some number-theoretic algorithms (hence the name), but since then it grew to include many different little utility functions. For example, CRT-reconstruction of polynomials in coefficient representation, conversion functions between different types, procedures to sample at random from various distributions, etc.

2.3 bluestein and Cmodulus: Polynomials in FFT Representation

The bluestein module implements a non-power-of-two FFT over a prime field $\mathbb{Z}_p$, using the Bluestein FFT algorithm [1]. We use modulo-p polynomials to encode the FFTs inputs and outputs. Specifically this module builds on Shoup's NTL library [9], and contains both a bigint version with types `ZZ_p` and `ZZ_pX`, and a smallint version with types `zz_p` and `zz_pX`. We have the following functions:

```
void BluesteinFFT(ZZ_pX& x, long n,
                  const ZZ_p& root, ZZ_pX& powers,
                  Vec<mulmod_precon_t>& powers_aux,
                  FFTRep& Rb, fftrep_aux& Rb_aux, FFTRep& Ra);

void BluesteinFFT(zz_pX& x, long n,
                  const zz_p& root, zz_pX& powers,
                  Vec<mulmod_precon_t>& powers_aux,
                  fftRep& Rb, fftrep_aux& Rb_aux, fftRep& Ra);
```

These functions compute length-n FFT of the coefficient-vector of `x` and put the result back in `x`. If the degree of `x` is less than n then it treats the top coefficients as 0, and if the degree is more than n then the extra coefficients are ignored. Similarly, if the top entries in the result `x` are zeros then `x` will have degree smaller than n. The argument `root` needs to be a *$2n$-th root of unity* in $\mathbb{Z}_p$. The inverse-FFT is obtained just by calling $\texttt{BluesteinFFT(...,root}^{-1}\texttt{,...)}$, but this procedure is *NOT SCALED*. Hence calling `BluesteinFFT(x,a,n,root,...)` and then $\texttt{BluesteinFFT(b,x,n,root}^{-1}\texttt{,...)}$ will result in having $\mathtt{b} = n \times \mathtt{a}$.

In addition to the size-n FFT of a which is returned in `x`, this procedure also returns the powers of `root` in the `powers` argument, $\texttt{powers} = \big(1, \texttt{root}, \texttt{root}^4, \texttt{root}^9, \ldots, \texttt{root}^{(n-1)^2}\big)$. In the `Rb` argument it returns the size-N FFT representation of the negative powers, for some $N \geq 2n-1$, N a power of two:

$$\textbf{Rb} \;=\; FFT_N\big(0,\ldots,0,\texttt{root}^{-(n-1)^2},\ldots,\texttt{root}^{-4},\texttt{root}^{-1},1,\texttt{root}^{-1},\texttt{root}^{-4},\ldots,\texttt{root}^{-(n-1)^2}0,\ldots,0\big).$$

On subsequent calls with the same `powers` and `Rb`, these arrays are not computed again but taken from the pre-computed arguments. If the `powers` and `Rb` arguments are initialized, then it is assumed that they were computed correctly from `root`. The behavior is undefined when calling with initialized `powers` and `Rb` but a different `root`. (In particular, to compute the inverse-FFT using $\texttt{root}^{-1}$, one must provide different `powers` and `Rb` arguments than those that were given when computing in the forward direction using `root`.)

The parameters `powers_aux` and `Rb_aux` are just used to store precomputed results depending on `powers` and Rb, respectively, from one call to the next: the caller should make sure that the same objects are consistently passed. The parameter `Ra` is just used for temporary storage, to minimize memory allocations.

The classes `Cmodulus` and `CModulus`. These classes provide an interface layer for the FFT routines above, relative to a single prime (where `Cmodulus` is used for smallint primes and `CModulus` for bigint primes). They keep the NTL "current modulus" structure for that prime, as well as the auxiliary structures `powers`, `Rb`, `powers_aux`, `Rb_aux`, and `Ra` arrays for FFT and inverse-FFT under that prime. They are constructed with the constructors

```
   Cmodulus(const PAlgebra& ZmStar, const long& q, const long& root);
   CModulus(const PAlgebra& ZmStar, const ZZ& q, const ZZ& root);
```

where `ZmStar` described the structure of $\mathbb{Z}_m^*$ (see Section 2.4), `q` is the prime modulus and `root` is a primitive $2m-$'th root of unity modulo q. (If the constructor is called with $\texttt{root} = 0$ then it computes a $2m$-th root of unity by itself.) Once an object of one of these classes is constructed, it provides an FFT interfaces via

```
   void Cmodulus::FFT(vec_long& y, const ZZX& x) const;  // y = FFT(x)
   void Cmodulus::iFFT(ZZX& x, const vec_long& y) const; // x = FFT^-1(y)
```

(And similarly for `CModulus` using `vec_ZZ` instead of `vec_long`). These method are inverses of each other.

2.4 PAlgebra: The Structure of $\mathbb{Z}_m^*$ and $\mathbb{Z}_m^*/\langle 2\rangle$

The class PAlgebra is the base class containing the structure of $\mathbb{Z}_m^*$, as well as the quotient group $\mathbb{Z}_m^*/\langle 2\rangle$. We represent $\mathbb{Z}_m^*$ as $\mathbb{Z}_m^* = \langle 2\rangle \times \langle g_1, g_2, \ldots\rangle \times \langle h_1, h_2, \ldots\rangle$, where each g_i has the same order in $\mathbb{Z}_m^*$ as in $\mathbb{Z}_m^*/\langle 2, g_1, \ldots, g_{i-1}\rangle$, and the h_i's generate the group $\mathbb{Z}_m^*/\langle 2, g_1, g_2, \ldots\rangle$ (and they do not have the same order in $\mathbb{Z}_m^*$ as in $\mathbb{Z}_m^*/\langle 2, g_1, \ldots\rangle$).

We compute this representation in a manner similar (but not identical) to the proof of the fundamental theorem of finitely generated abelian groups. Namely we keep the elements in equivalence classes of the "quotient group so far", and each class has a representative element (called a pivot), which in our case we just choose to be the smallest element in the class. Initially each element is in its own class. At every step, we choose the highest order element g in the current quotient group and add it as a new generator, then unify classes if their members are a factor of g from each other, repeating this process until no further unification is possible. Since we are interested in the quotient group $\mathbb{Z}_m^*/\langle 2\rangle$, we always choose 2 as the first generator.

One twist in this routine is that initially we only choose an element as a new generator if its order in the current quotient group is the same as in the original group $\mathbb{Z}_m^*$. Only after no such elements are available, do we begin to use generators that do not have the same order as in $\mathbb{Z}_m^*$.

Once we chose all the generators (and for each generator we compute its order in the quotient group where it was chosen), we compute a set of "slot representatives" as follows: Putting all the g_i's and h_i's in one list, let us denote the generators of $\mathbb{Z}_m^*/\langle 2\rangle$ by $\{f_1, f_2, \ldots, f_n\}$, and let $\mathsf{ord}(f_i)$ be the order of f_i in the quotient group at the time that it was added to the list of generators. The the slot-index representative set is

$$T \stackrel{\text{def}}{=} \left\{ \prod_{i=1}^{n} f_i^{e_i} \bmod m \ : \ \forall i, e_i \in \{0, 1, \ldots, \mathsf{ord}(f_i) - 1\} \right\}.$$

Clearly, we have $T \subset \mathbb{Z}_m^*$, and moreover T contains exactly one representative from each equivalence class of $\mathbb{Z}_m^*/\langle 2\rangle$. Recall that we use these representatives in our encoding of plaintext slots, where a polynomial $a \in \mathbb{A}_2$ is viewed as encoding the vector of $\mathbb{F}_{2^d}$ elements $\big(a(\rho^t) \in \mathbb{F}_{2^d} \ : \ t \in T\big)$, where ρ is some fixed primitive m-th root of unity in $\mathbb{F}_{2^d}$.

In addition to defining the sets of generators and representatives, the class PAlgebra also provides translation methods between representations, specifically:

`int ith_rep(unsigned i) const;`

Returns t_i, i.e., the i'th representative from T.

`int indexOfRep(unsigned t) const;`

Returns the index i such that $\texttt{ith\_rep}(i) = t$.

```
int exponentiate(const vector<unsigned>& exps,
    bool onlySameOrd=false) const;
```

Takes a vector of exponents, $(e_1, \ldots, e_n)$ and returns $t = \prod_{i=1}^{n} f_i^{e_i} \in T$.

`const int* dLog(unsigned t) const;`

On input some $t \in T$, returns the discrete-logarithm of t with the f_i's are bases. Namely, a vector `exps`$= (e_1, \ldots, e_n)$ such that `exponentiate(exps)`$= t$, and moreover $0 \le e_i \le \mathsf{ord}(f_i)$ for all i.

In fact, our implementation is slightly more general. The special generator 2 map be replaced with an arbitrary prime p. The constructor for the class PAlgebra is:

```
PAlgebra(unsigned m, unsigned p = 2);
```

Thus, a PAlgebra object is completely determined by m and p, where p defaults to $p = 2$.

2.5 PAlgebraMod: Plaintext Slots

This class implements the structure of the plaintext spaces, which is of the form $\mathbb{A}_{p^r} = \mathbb{A}/p^r\mathbb{A}$, for a small value of r. Recall that $\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X))$ and $\mathbb{A}_{p^r} = \mathbb{A}/(p^r) = \mathbb{Z}[X]/(\Phi_m(X), p^r)$. Typically, $r = 1$ for ordinary homomorphic computation, while we expect to use $r > 1$ for bootstrapping, as described in [6]. Also note that while, typically, $p = 2$ (the default value for p when constructing a PAlgebra object), an arbitrary prime p is allowed.

For the case $r = 1$, the plaintext slots are determined by the factorization of $\Phi_m(X)$ modulo p into ℓ degree-d polynomials. Once we have this factorization, $\Phi_m(X) = \prod_j F_j(X) \pmod{p}$, we choose an arbitrary factor as the "first factor", denote it $F_1(X)$, and this corresponds to the first input slot (whose representative is $1 \in T$).[1] With each representative $t \in T$ we then associate the factor $GCD(F_1(X^t), \Phi_m(X))$, with polynomial-GCD computed modulo p. Note that fixing a representation of the field $\mathbb{K} = \mathbb{Z}_p[X]/(F_1(X)) \cong \mathbb{F}_{p^d}$ and letting ρ be a root of F_1 in $\mathbb{K}$, we get that the factor associated with the representative t is the minimal polynomial of $\rho^{1/t}$. Yet another way of saying the same thing, if the roots of F_1 in $\mathbb{K}$ are

$$\rho, \rho^p, \rho^{p^2}, \ldots, \rho^{p^{d-1}},$$

then the roots of the factor associated to t are

$$\rho^{1/t}, \rho^{p/t}, \rho^{p^2/t}, \ldots, \rho^{p^{d-1}/t},$$

where the arithmetic in the exponent is modulo m.

For the case $r > 1$, we start with a factorization of $\Phi_m(X)$ modulo p as above, and then use Hensel lifting to obtain a factorization of $\Phi_m(X)$ modulo p^r. Note that even in the case where $r > 1$ and we set $\mathbb{K} = \mathbb{Z}_{p^r}[X]/(F_1)$, the very same correspondence among roots as noted in the previous paragraph still holds in this setting. Among other things, this correspondence guarantees that the various plaintext slot rings $\mathbb{Z}_{p^r}[X]/(F_i)$ are all isomorphic.

After computing the factors of $\Phi_m(X)$ modulo p^r and the correspondence between these factors and the representatives from T, the class PAlgebraMod provides encoding/decoding methods to pack elements in polynomials and unpack them back. This is really the main functionality provided by this class, and we shall proceed to describe the interface in some detail.

An object `alMod` of type PAlgebraMod holds the integer r, along with a PAlgebra object `zMStar`, which determines p and m. The object `alMod` also provides support for encoding and decoding plaintext slots, with respect to a specified plaintext slot subring as determined by a polynomial $G(X)$. If $r = 1$, then $G(X)$ may be any irreducible polynomial over $\mathbb{Z}_p$ whose degree divides d. (For example, for homomorphic AES we can use and d divisible by 8, and G will be the AES polynomial $G(X) = X^8 + X^4 + X^3 + X + 1$.)

For $r > 1$, our current implementation is more restrictive, requiring that either $\deg G(X) = 1$ or $G(X) = F_1$. With these constraints, we are ensured that the $\mathbb{Z}_{p^r}[X]/(G)$ corresponds to a subring of each of the plaintext slot rings $\mathbb{Z}_{p^r}[X]/(F_i)$ via an efficiently computable $\mathbb{Z}_{p^r}$-algebra isomorphism.

The PAlgebraMod object `alMod` stores various tables of objects in the polynomial ring $\mathbb{Z}_{p^r}[X]$. To do this most efficiently, if $p = 2$ and $r = 1$, then these polynomials are represented as NTL objects of type `GF2X`, and otherwise of type `zz_pX`. Thus, the types of these objects are not determined until run time. As such, we use a class hierarchy, as follows.

[1] In the current implementation, a natural lexicographic order is imposed on polynomials modulo p, and the smallest such factor with respect to this order is chosen as F_1. This ensures interoperability across different platforms.

- PAlgebraModBase is a virtual base class
- PAlegbraModDerived<type> is a derived template class, where type is either PA_GF2 or PA_zz_p.

  These latter two classes define various types representing polynomials, vectors, and other objects related to the choice of underlying representation, all defined via typedef's.
- The class PAlgebraMod itself is a simple wrapper around a smart pointer to a PAlgebraModBase object: copying a PAlgebraMod object results is a "deep copy" of the underlying object of the derived class.

The slot encoding/decoding operations cannot be performed directly using a PAlgebraMod object, but rather, they must be performed using objects of the derived class. So, given an object of type PAlgebraMod, which contains a pointer to an object of type PAlgebraModBase, one must first "downcast" this pointer to a pointer of type PAlgebraModDerived<type>. For convenience, the class PAlgebraMod provides a special "downcast" method to simplify the notation.

As discussed above, slot encoding and decoding is done with respect to a slot subring defined by a polynomial $G \in \mathbb{Z}_{p^r}[X]$. A call to the following method of PAlgebraModDerived<type> will precompute relevant data determined by G:

```
void mapToSlots(MappingData<type>& mappingData, const RX& G) const;
```

The precomputed data is stored in the parameter mappingData, which is a template type, parameterized in the same way as PAlgebraModDerived. Also, RX is a typedef that is defined as either GF2X or zz_pX, depending on the class parameter type.

A call to the the following method of PAlegbraModDerived<type> will create a plaintext (which is a polynomial modulo p^r) by embedding a vector of polynomials (modulo G) into the corresponding slots:

```
void embedInSlots(RX& H, const vector<RX>& alphas,
                  const MappingData<type>& mappingData) const;
```

Here, mappingData holds the precomputed data obtained from mapToSlots, alphas is the vector of slot values, and the resulting plaintext is stored in H.

The following method of PAlegbraModDerived<type> does the same thing, but embeds the same value alpha in each slot:

```
void embedInAllSlots(RX& H, const RX& alpha,
                     const MappingData<type>& mappingData) const;
```

The following method of PAlegbraModDerived<type> reverses the process, unpacking the slots of a given plaintext ptxt into a vector alphas:

```
void decodePlaintext(vector<RX>& alphas, const RX& ptxt,
     const MappingData<type>& mappingData) const;
```

Usage of the PAlgebraMod classes is similar to the higher-level EncryptedArray classes, as illustrated in Section 5.1.

Other functionality

Besides slot encoding/decoding, the class `PAlgebraMod` provides some other functionality as well.

First, a `PAlgebraMod` object stores some "mask tables" which is used by the class `EncryptedArray` — these tables help facilitate rotating the slots of encrypted data.

Second, a `PAlgebraMod` object may be used to to calculate the "linearized polynomial" associated to a $\mathbb{Z}_{p^r}$-linear map over $\mathbb{Z}_{p^r}[X]/(G)$. The interface here is similar to the slot encoding/decoding interface, in that the operations are performed using a `PAlgebraModDerived` object and a `MappingData` object defined by G. The following `PAlgebraModDerived`-method

```
void buildLinPolyCoeffs(vector<RX>& C, const vector<RX>& L,
                        const MappingData<type>& mappingData) const;
```

will compute the coefficient vector C corresponding to a linear map M described using L by its action on the power basis for $\mathbb{Z}_{p^r}[X]/(G)$; that is, $M(x^j \bmod G) = (L[j] \bmod G)$, for $j = 0 \ldots d-1$. The result is a coefficient vector C for the linearized polynomial representing M: for $h \in Z_{p^r}[X]$ of degree $< d$, $M(h(X) \bmod G) = \sum_{i=0}^{d-1}(C[j] \mod G) \cdot (h(X^{p^j}) \mod G)$.

Note that in the case where $r = 1$, such linearized polynomials are guaranteed to exist, according to the standard theory of finite fields. One can also show that such linearized polynomials exist also in the case $r > 1$, since C is the solution to a linear system of equations, and so it exists and be computed by Hensel lifting from a solution modulo p. (We remark that the method `buildLinPolyCoeffs` is used by `EncryptedArray` class, that that class also provides a higher-level interface.)

2.6 IndexSet and IndexMap: Sets and Indexes

In our implementation, all the polynomials are represented in double-CRT format, relative to some subset of the small primes in our list (cf. Section 1.2). The subset itself keeps changing throughout the computation, and so we can have the same polynomial represented at one point relative to many primes, then a small number of primes, then many primes again, etc. (For example see the implementation of key-switching in Section 3.1.6.) To provide flexibility with these transformations, the IndexSet class implements an arbitrary subset of nonnegative integers, and the IndexMap class implements a collection of data items that are indexed by such a subset.

2.6.1 The IndexSet class

The IndexSet class implements all the standard interfaces of the abstract data-type of a set, along with just a few extra interfaces that are specialized to sets of *nonnegative integers*. It uses the standard C++ container `vector<bool>` to keep the actual set, and provides the following methods:

Constructors. The constructors `IndexSet()`, `IndexSet(long j)`, and `IndexSet(long low, long high)`, initialize an empty set, a singleton, and an interval, respectively.

Empty sets and cardinality. The static method IndexSet`::emptySet()` provides a read-only access to an empty set, and the method `s.clear()` removes all the elements in `s`, which is equivalent to `s`=IndexSet`::emptySet()`.

The method `s.card()` returns the number of elements in `s`.

Traversing a set. The methods `s.first()` and `s.last()` return the smallest and largest element in the set, respectively. For an empty set `s`, `s.first()` returns 0 and `s.last()` returns -1.

The method `s.next(`j`)` return the next element after j, if any; otherwise $j+1$. Similarly `s.prev(`j`)` return the previous element before j, if any; otherwise $j-1$. With these methods, we can iterate through a set `s` using one of:

```
for (long i = s.first(); i <= s.last(); i = s.next(i)) ...
for (long i = s.last(); i >= s.first(); i = s.prev(i)) ...
```

Comparison and membership methods. `operator==` and `operator!=` are provided to test for equality, whereas `s1.disjointFrom(s2)` and its synonym `disjoint(s1,s2)` test if the two sets are disjoint. Also, `s.contains(j)` returns `true` if `s` contains the element j, `s.contains(other)` returns `true` if `s` is a superset of `other`. For convenience, the operators `<=`, `<`, `>=` and `>` are also provided for testing the subset relation between sets.

Set operations. The method `s.insert(j)` inserts the integer j if it is not in `s`, and `s.remove(j)` removes it if it is there.

Similarly `s1.insert(s2)` returns in `s1` the union of the two sets, and `s1.remove(s2)` returns in `s1` the set difference `s1` $\setminus$ `s2`. Also, `s1.retain(s2)` returns in `s1` the intersection of the two sets. For convenience we also provide the operators `s1|s2` (union), `s1&s2` (intersection), `s1^s2` (symmetric difference, aka xor), and `s1/s2` (set difference).

2.6.2 The IndexMap class

The class template IndexMap<T> implements a map of elements of type `T`, indexed by a dynamic IndexSet. Additionally, it allows new elements of the map to be initialized in a flexible manner, by providing an initialization function which is called whenever a new element (indexed by a new index j) is added to the map.

Specifically, we have a helper class template `IndexMapInit<T>` that stores a pointer to an initialization function, and possibly also other parameters that the initialization function needs. We then provide a constructor

```
IndexMap(IndexMapInit<T>* initObject=NULL)
```

that associates the given initialization object with the new IndexMap object. Thereafter, when a new index j is added to the index set, an object `t` of type `T` is created using the default constructor for `T`, after which the function `initObject->init(t)` is called.

In our library, we use an IndexMap to store the rows of the matrix of a Double-CRT object. For these objects we have an initialization object that stores the value of $\phi(m)$, and the initialization function, which is called whenever we add a new row, ensures that all the rows have length exactly $\phi(m)$.

After initialization, an IndexMap object provides the operator `map[i]` to access the type-`T` object indexed by i (if i currently belongs to the IndexSet), as well as the methods `map.insert(i)` and `map.remove(i)` to insert or delete a single data item indexed by i, and also `map.insert(s)` and `map.remove(s)` to insert or delete a collection of data items indexed by the IndexSet `s`.

2.7 FHEcontext: Keeping the parameters

Objects in higher layers of our library are defined relative to some parameters, such as the integer parameter m (that defines the groups $\mathbb{Z}_m^*$ and $\mathbb{Z}_m^*/\langle p\rangle$ and the ring $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$) and the sequence of small primes that determine our modulus-chain. To allow convenient access to these parameters, we define the class FHEcontext that keeps them all and provides access methods and some utility functions.

One thing that's included in FHEcontext is a vector of Cmodulus objects, holding the small primes that define our modulus chain:

```
vector<Cmodulus> moduli;  // Cmodulus objects for the different primes
```

We provide access to the Cmodulus objects via `context.ithModulus(i)` (that returns a reference of type `const Cmodulus&`), and to the small primes themselves via `context.ithPrime(i)` (that returns a `long`). The FHEcontext includes also the various algebraic structures for plaintext arithmetic, specifically we have the three data members:

`PAlgebra zMstar;     // The structure of` $\mathbb{Z}_m^*/\langle p\rangle$
`PAlgebraMod alMod;  // The structure of` $\mathbb{Z}[X]/(\Phi_m(X), p^r)$

In addition to the above, the FHEcontext contains a few IndexSet objects, describing various partitions of the index-set in the vector of moduli. These partitions are used when generating the key-switching matrices in the public key, and when using them to actually perform key-switching on ciphertexts.

One such partition is "ciphertext" vs. "special" primes: Freshly encrypted ciphertexts are encrypted relative to a subset of the small primes, called the *ciphertext primes*. All other primes are only used during key-switching, these are called the *special primes*. The ciphertext primes, in turn, are sometimes partitioned further into a number of "digits", corresponding to the columns in our key-switching matrices. (See the explanation of this partition in Section 3.1.6.) These subsets are stored in the following data members:

```
IndexSet ctxtPrimes;     // the ciphertext primes
IndexSet specialPrimes;  // the "special" primes
vector<IndexSet> digits; // digits of ctxt/columns of key-switching matrix
```

The FHEcontext class provides also some convenience functions for computing the product of a subset of small primes, as well as the "size" of that product (i.e., its logarithm), via the methods:

```
ZZ productOfPrimes(const IndexSet& s) const;
void productOfPrimes(ZZ& p, const IndexSet& s) const;

double logOfPrime(unsigned i) const;   // = log(ithPrime(i))
double logOfProduct(const IndexSet& s) const;
```

Finally, the FHEcontext module includes some utility functions for choosing parameters and adding moduli to the chain. The method `addPrime(long p, bool isSpecial)` adds a single prime p (either "special" or not), after checking that p has $2m$'th roots of unity and it is not already in the list. Then we have three higher-level functions:

`double AddPrimesBySize(FHEcontext& c, double size, bool special=false);`

Adds to the chain primes whose product is at least exp(`size`), returns the natural logarithm of the product of all added primes.

`double AddPrimesByNumber(FHEcontext& c, long n, long atLeast=1, bool special=false);`

Adds n primes to the chain, all at least as large as the `atLeast` argument, returns the natural logarithm of the product of all added primes.

`void buildModChain(FHEcontext& c, long d, long t=3);`

Build modulus chain for a circuit of depth d, using t digits in key-switching. This function puts d ciphertext primes in the `moduli` vector, and then as many "special" primes as needed to mod-switch fresh ciphertexts (see Section 3.1.6).

We also have a helper function that uses some pre-computed table to help select the values for the parameter m (defining the m'th cyclotomic ring):

`long FindM(long k, long L, long c, long p, long d, long s, long chosen_m, bool verbose=false);`

In this method, `k` is the security parameter, `L` is the number of ciphertext-primes that we want to support, `c` is the number of columns in our key-switching matrices. The arguments `p, d` determine the plaintext space $\mathbb{F}_{p^d}$, the argument `s` bounds from below the number of plaintext slots that we want to support, and `chosen_m` gives the ability to specify a particular m parameter and test if it satisfies all our constraints.

2.8 DoubleCRT: Efficient Polynomial Arithmetic

The heart of our library is the DoubleCRT class that manipulates polynomials in Double-CRT representation. A DoubleCRT object is tied to a specific FHEcontext, and at any given time it is defined relative to a subset of small primes from our list, $S \subseteq [0, \ldots, \texttt{context.moduli.size}() - 1]$. Denoting the product of these small primes by $q = \prod_{i \in S} p_i$, a DoubleCRT object represents a polynomial $a \in \mathbb{A}_q$ by a matrix with $\phi(m)$ columns and one row for each small prime p_i (with $i \in S$). The i'th row contains the FFT representation of a modulo p_i, i.e., the evaluations $\{[a(\zeta_i{}^j)]_{p_i} : j \in \mathbb{Z}_m^*\}$, where ζ_i is some primitive m-th root of unity modulo p_i.

Although the FHEcontext must remain fixed throughout, the set S of primes can change dynamically, and so the matrix can lose some rows and add other ones as we go. We thus keep these rows in a dynamic IndexMap data member, and the current set of indexes S is available via the method `getIndexSet()`. We provide the following methods for changing the set of primes:

`void addPrimes(const IndexSet& s);`

Expand the index set by `s`. It is assumed that `s` is disjoint from the current index set. This is an expensive operation, as it needs to convert to coefficient representation and back, in order to determine the values in the added rows.

`double addPrimesAndScale(const IndexSet& S);`

Expand the index set by `S`, and multiply by $q_{\text{diff}} = \prod_{i \in S} p_i$. The set `S` is assumed to be disjoint from the current index set. Returns $\log(q_{\text{diff}})$. This operation is typically much faster than `addPrimes`, since we can fill the added rows with zeros.

`void removePrimes(const IndexSet& s);`

Remove the primes p_i with $i \in \mathtt{s}$ from the current index set.

`void scaleDownToSet(const IndexSet& s, long ptxtSpace);`

This is a modulus-switching operation. Let Δ be the set of primes that are removed, $\Delta = \mathtt{getIndexSet}() \setminus \mathtt{s}$, and $q_{\rm diff} = \prod_{i\in\Delta} p_i$. This operation removes the primes $p_i, i \in \Delta$, scales down the polynomial by a factor of $q_{\rm diff}$, and rounds so as to keep a mod `ptxtSpace` unchanged.

We provide some conversion routines to convert polynomials from coefficient-representation (NTL's `ZZX` format) to DoubleCRT and back, using the constructor

```
DoubleCRT(const ZZX&, const FHEcontext&, const IndexSet&);
```

and the conversion function `ZZX to_ZZX(const DoubleCRT&)`.

We support the usual set of arithmetic operations on DoubleCRT objects (e.g., addition, multiplication, etc.), always working in $\mathbb{A}_q$ for some modulus q. We only implemented the "destructive" two-argument version of these operations, where one of the input arguments is modified to return the result. These arithmetic operations can only be applied to DoubleCRT objects relative to the same FHEcontext, else an error is raised.

The DoubleCRT class supports operations between objects with different IndexSet's, offering two options to resolve the differences: our arithmetic operations take a boolean flag `matchIndexSets`; when the flag is set to *true* (which is the default), the index-set of the result is the union of the index-sets of the two arguments. When `matchIndexSets`=*false* then the index-set of the result is the same as the index-set of `*this`, i.e., the argument that will contain the result when the operation ends. The option `matchIndexSets`=*true* is slower, since it may require adding primes to the two arguments. Below is a list of the arithmetic routines that we implemented:

```
DoubleCRT& Negate(const DoubleCRT& other); // *this = -other
DoubleCRT& Negate();                       // *this = -*this;

DoubleCRT& operator+=(const DoubleCRT &other); // Addition
DoubleCRT& operator+=(const ZZX &poly); // expensive
DoubleCRT& operator+=(const ZZ &num);
DoubleCRT& operator+=(long num);

DoubleCRT& operator-=(const DoubleCRT &other); // Subtraction
DoubleCRT& operator-=(const ZZX &poly); // expensive
DoubleCRT& operator-=(const ZZ &num);
DoubleCRT& operator-=(long num);

// These are the prefix versions, ++dcrt and --dcrt.
DoubleCRT& operator++();
DoubleCRT& operator--();

// Postfix versions (return type is void, it is offered just for style)
void operator++(int);
void operator--(int);
```

```
  DoubleCRT& operator*=(const DoubleCRT &other); // Multiplication
  DoubleCRT& operator*=(const ZZX &poly); // expensive
  DoubleCRT& operator*=(const ZZ &num);
  DoubleCRT& operator*=(long num);

  // Procedural equivalents, providing also the matchIndexSets flag

  void Add(const DoubleCRT &other, bool matchIndexSets=true);
  void Sub(const DoubleCRT &other, bool matchIndexSets=true);
  void Mul(const DoubleCRT &other, bool matchIndexSets=true);

  DoubleCRT& operator/=(const ZZ &num);  // Division by constant
  DoubleCRT& operator/=(long num);

  void Exp(long k);    // Small-exponent polynomial exponentiation

  // Automorphism F(X) --> F(X^k)  (with gcd(k,m)==1)
  void automorph(long k);
  DoubleCRT& operator>>=(long k);
```

We also provide methods for choosing at random polynomials in DoubleCRT format, as follows:

`void randomize(const ZZ* seed=NULL);`
: Fills each row $i \in$ `getIndexSet()` with random integers modulo p_i. This procedure uses the NTL PRG, setting the seed to the `seed` argument if it is non-NULL, and using the current PRG state of NTL otherwise.

`void sampleSmall();`
: Draws a random polynomial in coefficient representation, and converts it to DoubleCRT format. Each coefficient is chosen as 0 with probability 1/2, and as ± 1 with probability 1/4 each.

`void sampleHWt(long weight);`
: Draws a random polynomial with coefficients $-1, 0, 1$, and converts it to DoubleCRT format. The polynomial is chosen at random subject to the condition that all but `weight` of its coefficients are zero, and the non-zero coefficients are random in ± 1.

`void sampleGaussian(double stdev=3.2);`
: Draws a random polynomial with coefficients $-1, 0, 1$, and converts it to DoubleCRT format. Each coefficient is chosen at random from a Gaussian distribution with zero mean and variance $\texttt{stdev}^2$, rounded to an integer.

In addition to the above, we also provide the following methods:

```
  DoubleCRT& SetZero(); // set to the constant zero
  DoubleCRT& SetOne();  // set to the constant one
  const FHEcontext& getContext() const; // access to context
  const IndexSet& getIndexSet() const;  // the current set of primes

  void breakIntoDigits(vector<DoubleCRT>&, long) const;
  // used in key-switching
```

The method `breakIntoDigits` above is described in Section 3.1.6, where we discuss key-switching.

The SingleCRT class. SingleCRT is a helper class, used to gain some efficiency in expensive DoubleCRT operations. A SingleCRT object is also defined relative to a fixed FHEcontext and a dynamic subset S of the small primes. This SingleCRT object holds an IndexMap of polynomials (in NTL's ZZX format), where the i'th polynomial contains the coefficients modulo the ith small prime in our list.

Although SingleCRT and DoubleCRT objects can interact in principle, translation back and forth are expensive since they involve FFT (or inverse FFT) modulo each of the primes. Hence support for interaction between them is limited to explicit conversions.

3 The Crypto Layer

The third layer of our library contains the implementation of the actual BGV homomorphic cryptosystem, supporting homomorphic operations on the "native plaintext space" $\mathbb{A}_{p^r}$. We partitioned this layer (somewhat arbitrarily) into the `Ctxt` module that implements ciphertexts and ciphertext arithmetic, the `FHE` module that implements the public and secret keys, and the key-switching matrices, and a helper KeySwitching module that implements some common strategies for deciding what key-switching matrices to generate. Two high-level design choices that we made in this layer were to implement ciphertexts as arbitrary-length vectors of polynomials, and to allow more than one secret-key per instance of the system. These two choices are described in more detail in Sections 3.1 and 3.2 below, respectively.

3.1 The Ctxt module: Ciphertexts and homomorphic operations

Recall that in the BGV cryptosystem, a "canonical" ciphertext relative to secret key $\mathfrak{s} \in \mathbb{A}$ is a pair polynomials $(\mathfrak{c}_0, \mathfrak{c}_1) \in \mathbb{A}_q \times \mathbb{A}_q$ (for the "current modulus" q), such that $\mathfrak{m} = [\mathfrak{c}_0 + \mathfrak{c}_1\mathfrak{s}]_q$ is a polynomial with small coefficients, and the plaintext that is encrypted by this ciphertext is the polynomial $[\mathfrak{m}]_{p^r} \in \mathbb{A}_{p^r}$. However the library has to deal also with "non-canonical" ciphertexts: for example when multiplying two ciphertexts as above we get a vector of three polynomials $(\mathfrak{c}_0, \mathfrak{c}_1, \mathfrak{c}_2)$, which is decrypted by setting $\mathfrak{m} = [\mathfrak{c}_0 + \mathfrak{c}_1\mathfrak{s} + \mathfrak{c}_2\mathfrak{s}^2]_q$ and outputting $[\mathfrak{m}]_{p_r}$. Also, after a homomorphic automorphism operation we get a two-polynomial ciphertext $(\mathfrak{c}_0, \mathfrak{c}_1)$ but relative to the key $\mathfrak{s}' = \kappa(\mathfrak{s})$ (where κ is the same automorphism that we applied to the ciphertext, namely $\mathfrak{s}'(X) = \mathfrak{s}(X^t)$ for some $t \in \mathbb{Z}_m^*$).

To support all of these options, a ciphertext in our library consists of an arbitrary-length vector of "ciphertext parts", where each part is a polynomial, along with a "handle" that points to the secret-key that should multiply this part at decryption. Handles, parts, and ciphertexts are implemented using the classes SKHandle, CtxtPart, and Ctxt, respectively.

3.1.1 The SKHandle class

An object of the SKHandle class "points" to one particular secret-key polynomial, that should multiply one ciphertext-part upon decryption. Recall that we allow multiple secret keys per instance of the cryptosystem, and that we may need to reference powers of these secret keys (e.g. $\mathfrak{s}^2$ after multiplication) or polynomials of the form $\mathfrak{s}(X^t)$ (after automorphism). The general form of these secret-key polynomials is therefore $\mathfrak{s}_i^r(X^t)$, where $\mathfrak{s}_i$ is one of the secret keys associated with this instance, r is the power of that secret key, and t is the automorphism that we applied to it. To uniquely identify a single secret-key polynomial that should be used during decryption, we should therefore keep the three integers (i, r, t).

Accordingly, a SKHandle object has three integer data members, `powerOfS`, `powerOfX`, and `secretKeyID`. It is considered a reference to the constant polynomial 1 whenever `powerOfS`$= 0$, irrespective of the other two values. Also, we say that a SKHandle object points to a *base secret key* if it has `powerOfS` $=$ `powerOfX` $= 1$.

Observe that when multiplying two ciphertext parts, we get a new ciphertext part that should be multiplied upon decryption by the product of the two secret-key polynomials. This gives us the following set of rules for multiplying SKHandle objects (i.e., computing the handle of the resulting ciphertext-part after multiplication). Let $\{\texttt{powerOfS}, \texttt{powerOfX}, \texttt{secretKeyID}\}$ and $\{\texttt{powerOfS}', \texttt{powerOfX}', \texttt{secretKeyID}'\}$ be two handles to be multiplied, then we have the following rules:

- If one of the SKHandle objects points to the constant 1, then the result is equal to the other one.
- If neither points to one, then we must have $\texttt{secretKeyID} = \texttt{secretKeyID}'$ and $\texttt{powerOfX} = \texttt{powerOfX}'$, otherwise we cannot multiply. If we do have these two equalities, then the result will also have the same $t = \texttt{powerOfX}$ and $i = \texttt{secretKeyID}$, and it will have $r = \texttt{powerOfS} + \texttt{powerOfS}'$.

The methods provided by the SKHandle class are the following:

```
SKHandle(long powerS=0, long powerX=1, long sKeyID=0); // constructor
long getPowerOfS() const;    // returns powerOfS;
long getPowerOfX() const;    // returns powerOfX;
long getSecretKeyID() const; // return secretKeyID;

void setBase(); // set to point to a base secret key
void setOne();  // set to point to the constant 1
bool isBase() const; // does it point to base?
bool isOne() const;  // does it point to 1?

bool operator==(const SKHandle& other) const;
bool operator!=(const SKHandle& other) const;

bool mul(const SKHandle& a, const SKHandle& b);    // multiply the handles
    // result returned in *this, returns true if handles can be multiplied
```

3.1.2 The CtxtPart class

A ciphertext-part is a polynomial with a handle (that "points" to a secret-key polynomial). Accordingly, the class CtxtPart is derived from DoubleCRT, and includes an additional data member of type SKHandle. This class does not provide any methods beyond the ones that are provided by the base class DoubleCRT, except for access to the secret-key handle (and constructors that initialize it).

3.1.3 The Ctxt class

A Ctxt object is always defined relative to a fixed public key and context, both must be supplied to the constructor and are fixed thereafter. As described above, a ciphertext contains a vector of parts, each part with its own handle. This type of representation is quite flexible, for example you can in principle add ciphertexts that are defined with respect to different keys, as follows:

- For parts of the two ciphertexts that point to the same secret-key polynomial (i.e., have the same handle), you just add the two DoubleCRT polynomials.
- Parts in one ciphertext that do not have a counterpart in the other ciphertext will just be included in the result intact.

For example, suppose that you wanted to add the following two ciphertexts. one "canonical" and the other after an automorphism $X \mapsto X^3$:

$$\begin{array}{rcl} \vec{c} & = & (\mathfrak{c}_0[i=0,r=0,t=0],\ \mathfrak{c}_1[i=0,r=1,t=1]) \\ \text{and } \vec{c}\,' & = & (\mathfrak{c}'_0[i=0,r=0,t=0],\ \mathfrak{c}'_3[i=0,r=1,t=3]). \end{array}$$

Adding these ciphertexts, we obtain a three-part ciphertext,

$$\vec{c}+\vec{c}\,' = ((\mathfrak{c}_0+\mathfrak{c}'_0)[i=0,r=0,t=0],\quad \mathfrak{c}_1[i=0,r=1,t=1],\quad \mathfrak{c}'_3[i=0,r=1,t=3]).$$

Similarly, we also have flexibility in multiplying ciphertexts using a tensor product, as long as all the pairwise handles of all the parts can be multiplied according to the rules from Section 3.1.1 above.

The Ctxt class therefore contains a data member `vector<CtxtPart> parts` that keeps all of the ciphertext-parts. By convention, the first part, `parts[0]`, always has a handle pointing to the constant polynomial 1. Also, we maintain the invariant that all the DoubleCRT objects in the parts of a ciphertext are defined relative to the same subset of primes, and the IndexSet for this subset is accessible as `ctxt.getPrimeSet()`. (The current BGV modulus for this ciphertext can be computed as $q =$ `ctxt.getContext().productOfPrimes(ctxt.getPrimeSet())`.)

A `Ctxt` object also contains a data member `ptxtSpace` that holds the plaintext-modulus for this ciphertext (i.e. the integer p such that the native plaintext space is $\mathbb{A}_p$, note that p is either a prime or a prime power). For reasons that will be discussed when we describe modulus-switching in Section 3.1.5, we maintain the invariant that a ciphertext relative to current modulus q and plaintext space p has an extra factor of $q \bmod p$. Namely, when we decrypt the ciphertext vector $\vec{c}$ using the secret-key vector $\vec{s}$, we compute $\mathfrak{m} = [\langle \vec{s}, \vec{c} \rangle]_p$ and then output the plaintext $m = [q^{-1} \cdot \mathfrak{m}]_p$ (rather than just $m = [\mathfrak{m}]_p$). Note that this has no effect when the plaintext space is $p = 2$, since $q^{-1} \bmod p = 1$ in this case.

The basic operations that we can apply to ciphertexts (beyond encryption and decryption) are addition, multiplication, addition and multiplication by constants, automorphism, key-switching and modulus switching. These operations are described in more detail later in this section.

3.1.4 Noise estimate

In addition to the vector of parts, a ciphertext contains also a heuristic estimate of the "noise variance", kept in the `noiseVar` data member: Consider the polynomial $\mathfrak{m} = [\langle \vec{c}, \vec{s} \rangle]_q$ that we obtain during decryption (before the reduction modulo 2). Thinking of the ciphertext $\vec{c}$ and the secret key $\vec{s}$ as random variables, this makes also $\mathfrak{m}$ a random variable. The data member `noiseVar` is intended as an estimate of the second moment of the random variable $\mathfrak{m}(\tau_m)$, where $\tau_m = e^{2\pi i/m}$ is the principal complex primitive m-th root of unity. Namely, we have $\texttt{noiseVar} \approx \mathsf{E}[|\mathfrak{m}(\tau_m)|^2] = \mathsf{E}[\mathfrak{m}(\tau_m) \cdot \overline{\mathfrak{m}(\tau_m)}]$, with $\overline{\mathfrak{m}(\tau_m)}$ the complex conjugate of $\mathfrak{m}(\tau_m)$.

The reason that we keep an estimate of this second moment, is that it gives a convenient handle on the l_2 canonical embedding norm of $\mathfrak{m}$, which is how we measure the noise magnitude in the ciphertext. Heuristically, the random variables $\mathfrak{m}(\tau_m^j)$ for all $j \in \mathbb{Z}_m^*$ behave as if they are identically distributed, hence the expected squared l_2 norm of the canonical embedding of $\mathfrak{m}$ is

$$\mathsf{E}\left[(\|\mathfrak{m}\|_2^{\text{canon}})^2\right] = \sum_{j \in \mathbb{Z}_m^*} \mathsf{E}\left[\left|\mathfrak{m}(\tau_m^j)\right|^2\right] \approx \phi(m) \cdot \mathsf{E}\left[\left|\mathfrak{m}(\tau_m)\right|^2\right] \approx \phi(m) \cdot \texttt{noiseVar}.$$

As the l_2 norm of the canonical embedding of $\mathfrak{m}$ is larger by a $\sqrt{\phi(m)}$ factor than the l_2 norm of its coefficient vector, we therefore use the condition $\sqrt{\texttt{noiseVar}} \geq q/2$ (with q the current BGV modulus) as our decryption-error condition. The library itself never checks this condition during the computation, but it provides a method `ctxt.isCorrect()` that the application can use to check for it. The library does use the noise estimate when deciding what primes to add/remove during modulus switching, see description of the `MultiplyBy` method below.

Recalling that the j'th ciphertext part has a handle pointing to some $\mathfrak{s}_j^{r_j}(X^{t_j})$, we have that $\mathfrak{m} = [\langle \vec{c}, \vec{s} \rangle]_q = [\sum_j \mathfrak{c}_j \mathfrak{s}_j^{r_j}]_q$. A valid ciphertext vector in the BGV cryptosystem can always be written as $\vec{c} = \vec{r} + \vec{e}$ such that $\vec{r}$ is some masking vector satisfying $[\langle \vec{r}, \vec{s} \rangle]_q = 0$ and $\vec{e} = (\mathfrak{e}_1, \mathfrak{e}_2, \ldots)$ is such that $\|\mathfrak{e}_j \cdot \mathfrak{s}_j^{r_j}(X^{t_j})\| \ll q$. We therefore have $\mathfrak{m} = [\langle \vec{c}, \vec{s} \rangle]_q = \sum_j \mathfrak{e}_j \mathfrak{s}_j^{r_j}$, and under the heuristic assumption that the "error terms" $\mathfrak{e}_j$ are independent of the keys we get $\mathsf{E}[|\mathfrak{m}(\tau_m)|^2] = \sum_j \mathsf{E}[|\mathfrak{e}_j(\tau_m)|^2] \cdot \mathsf{E}[|\mathfrak{s}_j(\tau_m^{t_j})^{r_j}|^2] \approx \sum_j \mathsf{E}[|\mathfrak{e}_j(\tau_m)|^2] \cdot \mathsf{E}[|\mathfrak{s}_j(\tau_m)^{r_j}|^2]$. The terms $\mathsf{E}[|\mathfrak{e}_j(\tau_m)|^2]$ depend on the the particular error polynomials that arise during the computation, and will be described when we discuss the specific operations. For the secret-key terms we use the estimate

$$\mathsf{E}\left[|\mathfrak{s}(\tau_m)^r|^2\right] \approx r! \cdot H^r,$$

where H is the Hamming-weight of the secret-key polynomial $\mathfrak{s}$. For $r = 1$, it is easy to see that $\mathsf{E}[|\mathfrak{s}(\tau_m)|^2] = H$: indeed, for every particular choice of the H nonzero coefficients of $\mathfrak{s}$, the random variable $\mathfrak{s}(\tau_m)$ (defined over the choice of each of these coefficients as ± 1) is a zero-mean complex random variable with variance exactly H (since it is a sum of exactly H random variables, each obtained by multiplying a uniform ± 1 by a complex constant of magnitude 1). For $r > 1$, it is clear that $\mathsf{E}[|\mathfrak{s}(\tau_m)^r|^2] \geq \mathsf{E}[|\mathfrak{s}(\tau_m)|^2]^r = H^r$, but the factor of $r!$ may not be clear. We first observed that factor experimentally, and then validated it analytically for the case $r \ll \min(\sqrt{H}, \log m)$ (which is what we use in our library). See details in the appendix. The rules that we use for computing and updating the data member `noiseVar` during the computation, as described below.

Encryption. For a fresh ciphertext, encrypted using the public encryption key, we have $\texttt{noiseVar} = \sigma^2(1 + \phi(m)^2/2 + \phi(m)(H + 1))$, where σ^2 is the variance in our RLWE instances, and H is the Hamming weight of the first secret key.

When the plaintext space modulus is $p > 2$, that quantity is larger by a factor of p^2. See Section 3.2.2 for the reason for these expressions.

Modulus-switching. The noise magnitude in the ciphertexts scales up as we add primes to the prime-set, while modulus-switching down involves both scaling down and adding a term (corresponding to the rounding errors for modulus-switching).

Namely, when adding more primes to the prime-set we scale up the noise estimate as $\texttt{noiseVar}' = \texttt{noiseVar} \cdot \Delta^2$, with Δ the product of the added primes. When removing primes from the prime-set we scale down and add an extra term, setting $\texttt{noiseVar}' = \texttt{noiseVar}/\Delta^2 + \texttt{addedNoise}$, where the added-noise term is computed as follows: We go over all the parts in the ciphertext, and consider their handles. For any part j with a handle that points to $\mathfrak{s}_j^{r_j}(X^{t_j})$, where $\mathfrak{s}_j$ is a secret-key polynomial whose coefficient vector has Hamming-weight H_j, we add a term $(p^2/12) \cdot \phi(m) \cdot (r_j)! \cdot H_j^{r_j}$. Namely, when modulus-switching down we set

$$\texttt{noiseVar}' = \texttt{noiseVar}/\Delta^2 + \sum_j \frac{p^2}{12} \cdot \phi(m) \cdot (r_j)! \cdot H_j^{r_j}.$$

See Section 3.1.5 for the reason for this expression.

Re-linearization/key-switching. When key-switching a ciphertext, we modulus-switch down to remove all the "special primes" from the prime-set of the ciphertext if needed (cf. Section 2.7). Then, the key-switching operation itself has the side-effect of adding these "special primes" back. These two modulus-switching operations have the effect of scaling the noise down, then back up, with the added noise term as above. Then add yet another noise term as follows:

The key-switching operation involves breaking the ciphertext into some number n' of "digits" (see Section 3.1.6). For each digit i of size D_i and every ciphertext-part that we need to switch (i.e., one that does not already point to 1 or a base secret key), we add a noise-term $\phi(m)\sigma^2 \cdot p^2 \cdot D_i^2/4$, where σ^2 is the variance in our RLWE instances. Namely, if we need to switch k parts and if $\texttt{noiseVar}'$ is the noise estimate after the modulus-switching down and up as above, then our final noise estimate after key-switching is

$$\texttt{noiseVar}'' = \texttt{noiseVar}' + k \cdot \phi(m)\sigma^2 \cdot p^2 \cdot \sum_{i=1}^{n'} D_i^2/4$$

where D_i is the size of the i'th digit. See Section 3.1.6 for more details.

addConstant. We roughly add the size of the constant to our noise estimate. The calling application can either specify the size of the constant, or else we use the default value $\texttt{sz} = \phi(m) \cdot (p/2)^2$. Recalling that when current modulus is q we need to scale up the constant by $q \bmod p$, we therefore set $\texttt{noiseVar}' = \texttt{noiseVar} + (q \bmod p)^2 \cdot \texttt{sz}$.

multByConstant. We multiply our noise estimate by the size of the constant. Again, the calling application can either specify the size of the constant, or else we use the default value $\texttt{sz} = \phi(m) \cdot (p/2)^2$. Then we set $\texttt{noiseVar}' = \texttt{noiseVar} \cdot \texttt{sz}$.

Addition. We first add primes to the prime-set of the two arguments until they are both defined relative to the same prime-set (i.e. the union of the prime-sets of both arguments). Then we just add the noise estimates of the two arguments, namely $\texttt{noiseVar}' = \texttt{noiseVar} + \texttt{other.noiseVar}$.

Multiplication. We first remove primes from the prime-set of the two arguments until they are both defined relative to the same prime-set (i.e. the intersection of the prime-sets of both arguments). Then the noise estimate is set to the product of the noise estimates of the two arguments, multiplied by an additional factor which is computed as follows: Let r_1 be the highest power of $\mathfrak{s}$ (i.e., the `powerOfS` value) in all the handles in the first ciphertext, and similarly let r_2 be the highest power of $\mathfrak{s}$ in all the handles in the second ciphertext, then the extra factor is $\binom{r_1+r_2}{r_1}$. Namely, we have $\texttt{noiseVar}' = \texttt{noiseVar} \cdot \texttt{other.noiseVar} \cdot \binom{r_1+r_2}{r_1}$. (In particular if the two arguments are canonical ciphertexts then the extra factor is $\binom{2}{1} = 2$.) See Section 3.1.7 for more details.

Automorphism. The noise estimate does not change by an automorphism operation.

3.1.5 Modulus-switching operations

Our library supports modulus-switching operations, both adding and removing small primes from the current prime-set of a ciphertext. In fact, our decision to include an extra factor of $(q \bmod p)$ in a ciphertext relative to current modulus q and plaintext-space modulus p, is mainly intended to somewhat simplify these operations.

To add primes, we just apply the operation `addPrimesAndScale` to all the ciphertext parts (which are polynomials in Double-CRT format). This has the effect of multiplying the ciphertext by the product of the added primes, which we denote here by Δ, and we recall that this operation is relatively cheap (as it involves no FFTs or CRTs, cf. Section 2.8). Denote the current modulus before the modulus-UP transformation by q, and the current modulus after the transformation by $q' = q \cdot \Delta$. If before the transformation we have $[\langle \vec{c}, \vec{s} \rangle]_q = \mathfrak{m}$, then after this transformation we have $\langle \vec{c}', \vec{s} \rangle = \langle \Delta \cdot \vec{c}, \vec{s} \rangle = \Delta \cdot \langle \vec{c}, \vec{s} \rangle$, and therefore $[\langle \vec{c}', \vec{s} \rangle]_{q\cdot\Delta} = \Delta \cdot \mathfrak{m}$. This means that if before the transformation we had by our invariant $[\langle \vec{c}, \vec{s} \rangle]_q = \mathfrak{m} \equiv q \cdot m \pmod{p}$, then after the transformation we have $[\langle \vec{c}, \vec{s} \rangle]_{q'} = \Delta \cdot \mathfrak{m} \equiv q' \cdot m \pmod{p}$, as needed.

For a modulus-DOWN operation (i.e., removing primes) from the current modulus q to the smaller modulus q', we need to scale the ciphertext $\vec{c}$ down by a factor of $\Delta = q/q'$ (thus getting a fractional ciphertext), then round appropriately to get back an integer ciphertext. Using our invariant about the extra factor of $(q \bmod p)$ in a ciphertext relative to modulus q (and plaintext space modulus p), we need to convert $\vec{c}$ into another ciphertext vector $\vec{c}'$ satisfying
(a) $(q')^{-1}\vec{c}' \equiv q^{-1}\vec{c} \pmod{p}$, and
(b) the "rounding error term" $\epsilon \overset{\text{def}}{=} \vec{c}' - (q'/q)\vec{c}$ is small.

As described in [5], we apply the following optimized procedure:

1. Let $\vec{\delta} = \vec{c} \bmod \Delta$,
2. Add or subtract multiples of Δ from the coefficients in $\vec{\delta}$ until it is divisible by p,
3. Set $\vec{c}^* = \vec{c} - \vec{\delta}$, // $\vec{c}^*$ divisible by Δ, and $\vec{c}^* \equiv \vec{c} \pmod{p}$
4. Output $\vec{c}' = \vec{c}/\Delta$.

An argument similar to the proof of [2, Lemma 4] shows that if before the transformation we had $\mathfrak{m} = [\langle \vec{c}, \vec{s} \rangle]_q \equiv q \cdot m \pmod{p}$, then after the transformation we have $\mathfrak{m}' = [\langle \vec{c}', \vec{s} \rangle]_{q'} \equiv q' \cdot m \pmod{p}$, as needed. (The difference from [2, Lemma 4] is that we do not assume that $q, q' \equiv 1 \pmod{p}$.)

Considering the noise magnitude, we can write $\vec{c}' = \vec{c}/\Delta + \vec{\epsilon}$ where $\vec{\epsilon}$ is the rounding error (i.e., the terms that are added in Step 2 above, divided by Δ). The noise polynomial is thus scaled down by a Δ factor, then increased by the additive term $a \overset{\text{def}}{=} \langle \vec{\epsilon}, \vec{s} \rangle = \sum_j \epsilon_j(X) \cdot \mathfrak{s}_j^{r_j}(X^{t_j})$ (with $a \in \mathbb{A}$). We make the heuristic assumption that the coefficients in all the ϵ_j's behave as if they are chosen uniformly in the interval $-[p/2, p/2)$. Under this assumption, we have

$$\mathsf{E}\left[|\epsilon_j(\tau_m)|^2\right] = \phi(m) \cdot p^2/12,$$

since the variance of a uniform random variable in $-[p/2, p/2)$ is $p^2/12$, and $\epsilon_j(\tau_m)$ is a sum of $\phi(m)$ such variables, scaled by different magnitude-1 complex constants. Assuming heuristically that the ϵ_j's are independent of the public key, we have

$$\mathsf{E}\left[|a(\tau_m)|^2\right] = \sum_j \mathsf{E}\left[|\epsilon_j(\rho_m)|^2\right] \cdot \mathsf{E}\left[\left|\mathfrak{s}_j^{r_j}(X^{t_j})\right|^2\right] \approx \sum_j (\phi(m) \cdot p^2/12) \cdot (r_j)! \cdot H_j^{r_j},$$

where p is the plaintext-space modulus, H_j is the Hamming weight of the secret key for the j'th part, and r_j is the power of that secret key.

3.1.6 Key-switching/re-linearization

The re-linearization operation ensures that all the ciphertext parts have handles that point to either the constant 1 or a base secret-key: Any ciphertext part j with a handle pointing to $\mathfrak{s}_j^{r_j}(X^{t_j})$ with either $r_j > 1$ or $r_j = 1$ and $t_j > 1$, is replace by two adding two parts, one that points to 1 and the other than points to $\mathfrak{s}_j(X)$, using some key-switching matrices from the public key. Also, a side-effect of re-linearization is that we add all the "special primes" to the prime-set of the resulting ciphertext.

To explain the re-linearization procedure, we begin by recalling that the "ciphertext primes" that define our moduli-chain are partitioned into some number $n \geq 1$ of "digits", of roughly equal size. (For example, say that we have 15 small primes in the chain and we partition them to three digits, then we may take the first five primes to be the first digit, the next five primes to be the second, and the last five primes to be the third.) The size of a digit is the product of all the primes that are associated with it, and below we denote by D_i the size of the i'th digit.

When key-switching a ciphertext $\vec{c}$ using $n > 1$ digits, we begin by breaking $\vec{c}$ into a collection of (at most) n lower-norm ciphertexts $\vec{c}_i$. First we remove all the special primes from the prime-set of the ciphertext by modulus-DOWN, if needed. Then we determine the smallest n' such that the product of of the first n' digits exceeds the current modulus q, and then we set

```
1. $\vec{d}_1 := \vec{c}$
2. For $i = 1$ to $n'$ do:
3.     $\vec{c}_i := \vec{d}_i \bmod D_i$            // $|\vec{c}_i| < D_i/2$
4.     $\vec{d}_{i+1} := (\vec{d}_i - \vec{c}_i)/D_i$     // $(\vec{d}_i - \vec{c}_i)$ divisible by $D_i$
```

Note that $\vec{c} = \sum_{i=1}^{n'} \vec{c}_i \cdot \prod_{j<i} D_i$, and also since $\|\vec{c}\|_\infty \le q/2 \le (\prod_i D_i)/2$, then it follows that $\|\vec{c}_i\|_\infty \le D_i/2$ for all i. Below we assume that the current modulus q is equal to the product of the first n' digits. (The case where $q < \prod_i D_i$ is very similar, but requires somewhat more complicated notations, in that case we just remove primes from the last digit until the product is equal to q.)

Consider now one particular ciphertext-part $\mathfrak{c}_j$ in $\vec{c}$, with a handle that points to some $\mathfrak{s}'_j(X) = \mathfrak{s}_j^{r_j}(X^{t_j})$, with either $r_j > 1$ or $r_j = 1$ and $t_j > 1$. Let us denote by $\mathfrak{c}_{i,j}$ the digit of $\mathfrak{c}_j$ in the low-norm ciphertext $\vec{c}_i$ from the procedure above. That is, we have $\mathfrak{c}_j = \sum_{i=1}^{n'} \mathfrak{c}_{i,j} \cdot \prod_{j<i} D_i$, and also $\|\mathfrak{c}_{i,j}\|_\infty \le D_i/2$ for all i. Moreover we need to have in the public-key a key-switching matrix for that handle, $W = W[\mathfrak{s}'_j \Rightarrow \mathfrak{s}_j]$. This W is a $2 \times n$ matrix of polynomials in Double-CRT format, defined relative to the product of all the small primes in our chain (special or otherwise). Below we denote the product of all these small primes by Q. The i'th column in the matrix encrypts the "plaintext polynomial" $\mathfrak{s}'_j \cdot \prod_{j<i} D_i$ under the key $\mathfrak{s}_j$, namely a vector $(\mathfrak{a}_i, \mathfrak{b}_i)^T \in A_Q^2$ such that $[\mathfrak{b}_i + \mathfrak{a}_i \mathfrak{s}_j]_Q = (\prod_{j<i} D_i) \cdot \mathfrak{s}'_j + p \cdot \mathfrak{e}_i$, for a small polynomial $\mathfrak{e}_i$ (and the plaintext-space modulus p). Moreover, as long as $(\prod_{j<i} D_i) \cdot \mathfrak{s}'_j + p \cdot \mathfrak{e}_i$ is short enough, the same equality holds also modulo smaller moduli that divide Q. In particular, denoting the product of the "special primes" by Q^* and the product of the n' digits that we use by q, then for all $i \le n'$ we have $\|(\prod_{j<i} D_i) \cdot \mathfrak{s}'_j + p \cdot \mathfrak{e}_i\|_\infty < qQ^*/2$, and therefore

$$\left[\mathfrak{b}_i + \mathfrak{a}_i \mathfrak{s}_j\right]_{qQ^*} = (\prod_{j<i} D_i) \cdot \mathfrak{s}'_j + p \cdot \mathfrak{e}_i.$$

We therefore reduce the key-switching matrix modulo qQ^*, and add the small primes corresponding to qQ^* to all the $\mathfrak{c}_{i,j}$'s. We replace $\mathfrak{c}_j$ by ciphertext-parts that point to 1 and base, by multiplying the n'-vector $\tilde{c}_j = (\mathfrak{c}_{1,j}, \ldots, \mathfrak{c}_{n',j})^T$ by (the first n' columns of) the key-switching matrix W, setting

$$(\mathfrak{c}''_j, \mathfrak{c}'_j)^T := \left[W[1:n'] \times \tilde{c}_j\right]_{qQ^*} \;=\; [\sum_{i=1}^{n'} (\mathfrak{a}_i, \mathfrak{b}_i)^T \cdot \mathfrak{c}_{i,j}]_{qQ^*}.$$

It is not hard to see that for these two new ciphertext-parts we have:

$$\begin{aligned}
\mathfrak{c}''_j + \mathfrak{c}'_j \mathfrak{s}_j &= \sum_{i=1}^{n'} (\mathfrak{b}_i + \mathfrak{a}_i \mathfrak{s}_j) \cdot \mathfrak{c}_{i,j} \;=\; \sum_{i=1}^{n'} \big((\prod_{j<i} D_i) \cdot \mathfrak{s}'_j + p \cdot \mathfrak{e}_i\big) \cdot \mathfrak{c}_{i,j} \\
&= \left(\sum_{i=1}^{n'} (\prod_{j<i} D_i) \mathfrak{c}_{i,j}\right) \mathfrak{s}'_j + p \cdot \sum_{i=1}^{n'} \mathfrak{e}_i \mathfrak{c}_{i,j} \;=\; \mathfrak{c}_j \mathfrak{s}'_j + p \sum_{i=1}^{n'} \mathfrak{e}_i \mathfrak{c}_{i,j} \pmod{qQ^*}
\end{aligned}$$

Replacing all the parts $\mathfrak{c}_j$ by pairs $(\mathfrak{c}''_j, \mathfrak{c}'_j)^T$ as above (and adding up all the parts that point to 1, as well as all the parts that point to the base $\mathfrak{s}_j$), we thus get a canonical ciphertext $\vec{c}' = (\tilde{\mathfrak{c}}_2, \tilde{\mathfrak{c}}_1)^T$, with $\tilde{\mathfrak{c}}_2 = [\sum_j \mathfrak{c}''_j]_{qQ^*}$ and $\tilde{\mathfrak{c}}_1 = [\sum_j \mathfrak{c}'_j]_{qQ^*}$, and we have

$$\tilde{\mathfrak{c}}_2 + \tilde{\mathfrak{c}}_1 \mathfrak{s}_j \;=\; \left(\sum_j \mathfrak{c}_j \mathfrak{s}'_j\right) + p\big(\sum_{i,j} \mathfrak{e}_i \mathfrak{c}_{i,j}\big) \;=\; \mathfrak{m} + p\big(\sum_{i,j} \mathfrak{e}_i \mathfrak{c}_{i,j}\big) \pmod{qQ^*}.$$

Hence, as long as the additive term $p(\sum_{i,j} \mathfrak{e}_i \mathfrak{c}_{i,j})$ is small enough, decrypting the new ciphertext yields the same plaintext value modulo p as decrypting the original ciphertext $\vec{c}$.

In terms of noise magnitude, we first scale up the noise by a factor of Q^* when adding all the special primes, and then add the extra noise term $p \cdot \sum_{i,j} \mathfrak{e}_i \mathfrak{c}_{i,j}$. Since the $\mathfrak{c}_{i,j}$'s have coefficients of magnitude at most $D_i/2$ and the polynomials $\mathfrak{e}_i$ are RLWE error terms with zero-mean coefficients and variance σ^2, then the second moment of $\mathfrak{e}_i(\tau_m) \cdot \mathfrak{c}_{i,j}(\tau_m)$ is no more than $\phi(m)\sigma^2 \cdot D_i^2/4$. Thus for every ciphertext part that we need to switch (i.e. that has a handle that points to something other than 1 or base), we add a term of $\phi(m)\sigma^2 \cdot p^2 \cdot D_i^2/4$. Therefore, if our noise estimate after the scale-up is $\texttt{noiseVar}'$ and we need to switch k

$$\texttt{noiseVar}'' = \texttt{noiseVar}' + k \cdot \phi(m)\sigma^2 \cdot p^2 \cdot \sum_{i=1}^{n'} D_i^2/4$$

3.1.7 Native arithmetic operations

The native arithmetic operations that can be performed on ciphertexts are negation, addition, subtraction, multiplication, addition of constants, multiplication by constant, and automorphism. In our library we expose to the application both the operations in their "raw form" without any additional modulus- or key-switching, as well as some higher-level interfaces for multiplication and automorphisms that include also modulus- and key-switching.

Negation. The method `Ctxt::negate()` transforms an encryption of a polynomial $m \in \mathbb{A}_p$ to an encryption of $[-m]_p$, simply by negating all the ciphertext parts modulo the current modulus. (Of course this has an effect on the plaintext only when $p > 2$.) The noise estimate is unaffected.

Addition/subtraction. Both of these operations are implemented by the single method

```
   void Ctxt::addCtxt(const Ctxt& other, bool negative=false);
```

depending on the `negative` boolean flag. For convenience, we provide the methods `Ctxt::operator+=` and `Ctxt::operator-=` that call `addCtxt` with the appropriate flag. A side effect of this operation is that the prime-set of `*this` is set to the union of the prime sets of both ciphertexts. After this scaling (if needed), every ciphertext-part in `other` that has a matching part in `*this` (i.e. a part with the same handle) is added to this matching part, and any part in `other` without a match is just appended to `*this`. We also add the noise estimate of both ciphertexts.

Constant addition. Implemented by the methods

```
   void Ctxt::addConstant(const ZZX& poly, double size=0.0);
   void Ctxt::addConstant(const DoubleCRT& poly, double size=0.0);
```

The constant is scaled by a factor $f = (q \bmod p)$, with q the current modulus and p the ciphertext modulus (to maintain our invariant that a ciphertext relative to q has this extra factor), then added to the part of `*this` that points to 1. The calling application can specify the size of the added constant (i.e. $|\texttt{poly}(\tau_m)|^2$), or else we use the default value $\texttt{size} = \phi(m) \cdot (p/2)^2$, and this value (times f^2) is added to our noise estimate.

Multiplication by constant. Implemented by the methods

```
   void Ctxt::multByConstant(const ZZX& poly, double size=0.0);
   void Ctxt::multByConstant(const DoubleCRT& poly, double size=0.0);
```

All the parts of `*this` are multiplied by the constant, and the noise estimate is multiplied by the size of the constant. As before, the application can specify the size, or else we use the default value $\mathsf{size} = \phi(m) \cdot (p/2)^2$.

Multiplication. "Raw" multiplication is implemented by

```
Ctxt& Ctxt::operator*=(const Ctxt& other);
```

If needed, we modulus-switch down to the intersection of the prime-sets of both arguments, then compute the tensor product of the two, namely the collection of all pairwise products of parts from `*this` and `other`.

The noise estimate of the result is the product of the noise estimates of the two arguments, times a factor which is computed as follows: Let r_1 be the highest power of $\mathfrak{s}$ (i.e., the `powerOfS` value) in all the handles in `*this`, and similarly let r_2 be the highest power of $\mathfrak{s}$ in all the handles `other`. The extra factor is then set as $\binom{r_1+r_2}{r_1}$. Namely, $\texttt{noiseVar}' = \texttt{noiseVar} \cdot \texttt{other.noiseVar} \cdot \binom{r_1+r_2}{r_1}$. The reason for the $\binom{r_1+r_2}{r_1}$ factor is that the ciphertext part in the result, obtained by multiplying the two parts with the highest `powerOfS` value, will have `powerOfS` value of the sum, $r = r_1 + r_2$. Recall from Section 3.1.4 that we estimate $\mathsf{E}[|\mathfrak{s}(\tau_m)^r|^2] \approx r! \cdot H^r$, where H is the Hamming weight of the coefficient-vector of $\mathfrak{s}$. Thus our noise estimate for the relevant part in `*this` is $r_1! \cdot H^{r_1}$ and the estimate for the part in `other` is $r_2! \cdot H^{r_2}$. To obtain the desired estimate of $(r_1 + r_2)! \cdot H^{r_1+r_2}$, we need to multiply the product of the estimates by the extra factor $\frac{(r_1+r_2)!}{r_1! \cdot r_2!} = \binom{r_1+r_2}{r_1}$. [2]

Higher-level multiplication. We also provide the higher-level methods

```
void Ctxt::multiplyBy(const Ctxt& other);
void Ctxt::multiplyBy(const Ctxt& other1, const Ctxt& other2);
```

The first method multiplies two ciphertexts, it begins by removing primes from the two arguments down to a level where the rounding-error from modulus-switching is the dominating noise term (see `findBaseSet` below), then it calls the low-level routine to compute the tensor product, and finally it calls the `reLinearize` method to get back a canonical ciphertext.

The second method that multiplies three ciphertexts also begins by removing primes from the two arguments down to a level where the rounding-error from modulus-switching is the dominating noise term. Based on the prime-sets of the three ciphertexts it chooses an order to multiply them (so that ciphertexts at higher levels are multiplied first). Then it calls the tensor-product routine to multiply the three arguments in order, and then re-linearizes the result.

We also provide the two convenience methods `square` and `cube` that call the above two-argument and three-argument multiplication routines, respectively.

Automorphism. "Raw" automorphism is implemented in the method

```
void Ctxt::automorph(long k);
```

For convenience we also provide `Ctxt& operator>>=(long k);` that does the same thing. These methods just apply the automorphism $X \mapsto X^k$ to every part of the current ciphertext, without changing the noise estimate, and multiply by k (modulo m) the `powerOfX` value in the handles of all these parts.

[2] Although products of other pairs of parts may need a smaller factor, the parts with highest `powerOfS` value represent the largest contribution to the overall noise, hence we use this largest factor for everything.

"Smart" Automorphism. Higher-level automorphism is implemented in the method
`void Ctxt::smartAutomorph(long k);`
The difference between `automorph` and `smartAutomorph` is that the latter ensures that the result can be re-linearized using key-switching matrices from the public key. Specifically, `smartAutomorph` breaks the automorphism $X \mapsto X^k$ into some number $t \geq 1$ of steps, $X \mapsto X^{k_i}$ for $i = 1, 2, \ldots t$, such that the public key contains key-switching matrices for re-linearizing all these steps (i.e. $W = W[\mathfrak{s}(X^{k_i}) \Rightarrow \mathfrak{s}(X)]$), and at the same time we have $\prod_{i=1}^{t} k_i = k \pmod{m}$. The method `smartAutomorph` begins by re-linearizing its argument, then in every step it performs one of the automorphisms $X \mapsto X^{k_i}$ followed by re-linearization.

The decision of how to break each exponent k into a sequence of k_i's as above is done off line during key-generation, as described in Section 3.2.2. After this off-line computation, the public key contains a table that for each $k \in \mathbb{Z}_m^*$ indicates what is the first step to take when implementing the automorphism $X \mapsto X^k$. The `smartAutomorph` looks up the first step k_1 in that table, performs the automorphism $X \mapsto X^{k_1}$, then compute $k' = k/k_1 \bmod m$ and does another lookup in the table for the first step relative to k', and so on.

3.1.8 More `Ctxt` methods

The `Ctxt` class also provides the following utility methods:

`void clear();` Removes all the parts and sets the noise estimate to zero.

`xdouble modSwitchAddedNoiseVar() const;` computes the added-noise from modulus-switching, namely it returns $\sum_j (\phi(m) \cdot p^2/12) \cdot (r_j)! \cdot H_j^{r_j}$ where H_j and r_j are respectively the Hamming weight of the secret key that the j'th ciphertext-part points to, and the power of that secret key (i.e., the `powerOfS` value in the relevant handle).

`void findBaseSet(IndexSet& s) const;` Returns in `s` the largest prime-set such that modulus-switching to `s` would make `ctxt.modSwitchAddedNoiseVar` the most significant noise term. In other words, modulus-switching to `s` results in a significantly smaller noise than to any larger prime-set, but modulus-switching further down would not reduce the noise by much. When multiplying ciphertexts using the `multiplyBy` "high-level" methods, the ciphertexts are reduced to (the intersection of) their "base sets" levels before multiplying.

`long getLevel() const;` Returns the number of primes in the result of `findBaseSet`.

`bool inCanonicalForm(long keyID=0) const;` Returns *true* if this is a canonical ciphertexts, with only two parts: one that points to 1 and the other that points to the "base" secret key $\mathfrak{s}_i(X)$, (where $i =$ `keyId` is specified by the caller).

`double log_of_ratio() const;` Returns $\log(\texttt{noiseVar})/2 - \log(q)$.

`bool isCorrect() const;` The method `isCorrect()` compares the noise estimate to the current modulus, and returns `true` if the noise estimate is less than half the modulus size. Specifically, if $\sqrt{\texttt{noiseVar}} < q/2$.

Access methods. Read-only access the data members of a `Ctxt` object:

```
const FHEcontext& getContext() const;
const FHEPubKey& getPubKey() const;
const IndexSet& getPrimeSet() const;
const xdouble& getNoiseVar() const;
const long getPtxtSpace() const; // the plaintext-space modulus
const long getKeyID() const;     // key-ID of the first part not pointing to 1
```

3.2 The FHE module: Keys and key-switching matrices

Recall that we made the high-level design choices to allow instances of the cryptosystem to have multiple secret keys. This decision was made to allow a *leveled* encryption system that does not rely on circular security, as well as to support switching to a different key for different purposes (which may be needed for bootstrapping, for example). However, we still view using just a single secret-key per instance (and relying on circular security) as the primary mode of using our library, and hence provided more facilities to support this mode than for the mode of using multiple keys. Regardless of how many secret keys we have per instance, there is always just a single public encryption key, for encryption relative to the first secret key. (The public key in our variant of the BGV cryptosystem is just a ciphertext, encrypting the constant 0.) In addition to this public encryption key, the public-key contains also key-switching matrices and some tables to help finding the right matrices to use in different settings. Ciphertexts relative to secret keys other than the first (if any), can only be generated using the key-switching matrices in the public key.

3.2.1 The KeySwitch class

This class implements key-switching matrices. As we described in Section 3.1.6, a key-switching matrix from $\mathfrak{s}'$ to $\mathfrak{s}$, denoted $W[\mathfrak{s}' \Rightarrow \mathfrak{s}]$, is a $2 \times n$ matrix of polynomials from $\mathbb{A}_Q$, where Q is the product of all the small primes in our chain (both ciphertext-primes and special-primes). Recall that the ciphertext primes are partitioned into n digits, where we denote the product of primes corresponding the i'th digit by D_i. Then the i'th column of the matrix $W[\mathfrak{s}' \Rightarrow \mathfrak{s}]$ is a pair of elements $(\mathfrak{a}_i, \mathfrak{b}_i) \in \mathbb{A}_Q^2$ that satisfy

$$[\mathfrak{b}_i + \mathfrak{a}_i \cdot \mathfrak{s}]_Q = (\prod_{j<i} D_j) \cdot \mathfrak{s}' + p \cdot \mathfrak{e}_i,$$

for a low-norm polynomial $\mathfrak{e}_i \in \mathbb{A}_Q$. In more detail, we choose a low-norm polynomial $\mathfrak{e}_i \in \mathbb{A}_Q$, where each coefficient of $\mathfrak{e}_i$ is chosen from a discrete Gaussian over the integers with variance σ^2 (with σ a parameter, by default $\sigma = 3.2$). Then we choose a random polynomial $\mathfrak{a}_i \in_R \mathbb{A}_Q$ and set $\mathfrak{b}_i := \left[(\prod_{j<i} D_i) \cdot \mathfrak{s}' + p \cdot \mathfrak{e}_i - \mathfrak{a}_i \cdot \mathfrak{s}\right]_Q$.

The matrix W is stored in a KeySwitch object in a space-efficient manner: instead of storing the random polynomials $\mathfrak{a}_i$ themselves, we only store a seed for a pseudorandom-generator, from which all the $\mathfrak{a}_i$'s are derived. The $\mathfrak{b}_i$'s are stored explicitly, however. We note that this space-efficient representation requires that we assume hardness of our ring-LWE instances even when the seed for generating the random elements is known, but this seems like a reasonable assumption.

In our library, the source secret key $\mathfrak{s}'$ is of the form $\mathfrak{s}' = \mathfrak{s}_{i'}^r(X^t)$ (for some index i' and exponents r, t), but the target $\mathfrak{s}$ must be a "base" secret-key, i.e. $\mathfrak{s} = \mathfrak{s}_i(X)$ for some index i. The

KeySwitch object stores in addition to the matrix W also a secret-key handle (r, t, i') that identifies the source secret key, as well as the index i of the target secret key.

The KeySwitch class provides a method `NumCols()` that returns the number of columns in the matrix W. We maintain the invariant that all the key-switching matrices that are defined relative to some context have the same number of columns, which is also equal to the number of digits that are specified in the context.

3.2.2 The FHEPubKey class

An FHEPubKey object is defined relative to a fixed FHEcontext, which must be supplied to the constructor and cannot be changed later. An FHEPubKey includes the public encryption key (which is a ciphertext of type `Ctxt`), a vector of key-switching matrices (of type `KeySwitch`), and another data structure (called `keySwitchMap`) that is meant to help finding the right key-switching matrices to use for automorphisms (see a more detailed description below). In addition, for every secret key in this instance, the FHEPubKey object stores the Hamming weight of that key, i.e., the number of non-zero coefficients of the secret-key polynomial. (This last piece of information is used to compute the estimated noise in a ciphertext.) The FHEPubKey class provides an encryption method, and various methods to find and access key-switching matrices.

`long Encrypt(Ctxt& ciphertxt, const ZZX& plaintxt, long ptxtSpace=0) const;` This method returns in `ciphertxt` an encryption of the plaintext polynomial `plaintxt`, relative to the plaintext-space modulus given in `ptxtSpace`. If the `ptxtSpace` parameter is not specified then we use the plaintext-space modulus from the public encryption key in this FHEPubKey object. If `ptxtSpace` is specified then we use the greater common divisor (GCD) of the specified value and the one from the public encryption key. The current-modulus in the new fresh ciphertext is the product of all the ciphertext-primes in the context, which is the same as the current modulus in the public encryption key in this FHEPubKey object.

Let the public encryption key in the FHEPubKey object be denoted $\vec{c}^* = (\mathfrak{c}_0^*, \mathfrak{c}_1^*)$, let Q_{ct} be the product of all the ciphertext primes in the context, and let p be the plaintext-space modulus (namely the GCD of the parameter `ptxtSpace` and the plaintext-space modulus from the public encryption key). The `Encrypt` method chooses a random low-norm polynomial $\mathfrak{r} \in \mathbb{A}_{Q_{\mathsf{ct}}}$ with $-1/0/1$ coefficients, and low-norm error polynomials $\mathfrak{e}_0, \mathfrak{e}_1 \in \mathbb{A}_Q$, where each coefficient of $\mathfrak{e}_i$'s is chosen from a discrete Gaussian over the integers with variance σ^2 (with σ a parameter, by default $\sigma = 3.2$). We then compute and return the canonical ciphertext

$$\vec{c} = (\mathfrak{c}_0, \mathfrak{c}_1) \ := \ \mathfrak{r} \cdot (\mathfrak{c}_0^*, \mathfrak{c}_1^*) + p \cdot (\mathfrak{e}_0, \mathfrak{e}_1) + \texttt{plaintxt}.$$

Note that since the public encryption key satisfies $[\mathfrak{c}_0^* + \mathfrak{s} \cdot \mathfrak{c}_1^*]_{Q_{\mathsf{ct}}} = p \cdot \mathfrak{e}^*$ for some low-norm polynomial $\mathfrak{e}^*$, then we have

$$[\mathfrak{c}_0 + \mathfrak{s} \cdot \mathfrak{c}_1]_{Q_{\mathsf{ct}}} = [\mathfrak{r} \cdot (\mathfrak{c}_0^* + \mathfrak{s} \cdot \mathfrak{c}_1^*) + p \cdot (\mathfrak{e}_0 + \mathfrak{s} \cdot \mathfrak{e}_1) + \texttt{plaintxt}]_{Q_{\mathsf{ct}}} = p \cdot (\mathfrak{e}_0 + \mathfrak{s} \cdot \mathfrak{e}_1 + \mathfrak{r} \cdot \mathfrak{e}^*) + \texttt{plaintxt}.$$

For the noise estimate in the new ciphertext, we multiply the noise estimate in the public encryption key by the size of the low-norm $\mathfrak{r}$, and add another term for the expression $\mathfrak{a} = p \cdot (\mathfrak{e}_0 + \mathfrak{s} \cdot \mathfrak{e}_1) + \texttt{plaintxt}$. Specifically, the noise estimate in the public encryption key is `pubEncrKey.noiseVar` $= \phi(m)\sigma^2 p^2$, the second moment of $\mathfrak{r}(\tau_m)$ is $\phi(m)/2$, and the second moment of $a(\tau_m)$ is no more than

$p^2(1+\sigma 2^{\phi}(m)(H+1))$ with H the Hamming weight of the secret key $\mathfrak{s}$. Hence the noise estimate in a freshly encrypted ciphertext is

$$\texttt{noiseVar} \;=\; p^2 \cdot \big(1 + \;\sigma^2\phi(m)\cdot\big(\phi(m)/2 + H + 1\big)\big).$$

The key-switching matrices. An FHEPubKey object keeps a list of all the key-switching matrices that were generated during key-generation in the data member `keySwitching` of type `vector<KeySwitch>`. As explained above, each key-switching matrix is of the form $W[\mathfrak{s}_i^r(X^t) \Rightarrow \mathfrak{s}_j(X)]$, and is identified by a `SKHandle` object that specifies (r,t,i) and another integer that specifies the target key-ID j. The basic facility provided to find a key-switching matrix are the two equivalent methods

```
const KeySwitch& getKeySWmatrix(const SKHandle& from, long toID=0) const;
const KeySwitch& getKeySWmatrix(long fromSPower, long fromXPower,
                                long fromID=0, long toID=0) const;
```

These methods return either a read-only reference to the requested matrix if it exists, or otherwise a reference to a dummy `KeySwitch` object that has `toKeyID` $= -1$. For convenience we also provide the methods `bool haveKeySWmatrix` that only test for existence, but do not return the actual matrix. Another variant is the method

```
const KeySwitch& getAnyKeySWmatrix(const SKHandle& from) const;
```

(and its counterpart `bool haveAnyKeySWmatrix`) that look for a matrix with the given source (r,t,i) and any target. All these methods first try to find the requested matrix using the `keySwitchMap` table (which is described below), and failing that they resort to linear search through the entire list of matrices.

The `keySwitchMap` table. Although our library supports key-switching matrices of the general form $W[\mathfrak{s}_i^r(X^t) \Rightarrow \mathfrak{s}_j(X)]$, we provide more facilities for finding matrices to re-linearize after automorphism (i.e., matrices of the form $W[\mathfrak{s}_i(X^{t_i}) \Rightarrow \mathfrak{s}_i(X)]$) than for other types of matrices.

For every secret key $\mathfrak{s}_i$ in the current instance we consider a graph G_i over the vertex set $\mathbb{Z}_m^*$, where we have an edge $j \to k$ if and only if we have a key-switching matrix $W[\mathfrak{s}_i(X^{jk^{-1}}) \Rightarrow \mathfrak{s}_i(X)]$) (where jk^{-1} is computed modulo m). We observe that if the graph G_i has a path $t \rightsquigarrow 1$ then we can apply the automorphism $X \mapsto X^t$ with re-linearization to a canonical ciphertext relative to $\mathfrak{s}_i$ as follows: Denote the path from t to 1 in the graph by

$$t = k_1 \to k_2 \cdots k_n = 1.$$

We follow the path one step at a time, for each step j applying the automorphism $X \mapsto X^{k_j k_{j+1}^{-1}}$ and then re-linearizing the result using the matrix $W[\mathfrak{s}_i(X^{k_j k_{j+1}^{-1}}) \Rightarrow \mathfrak{s}_i(X)]$ from the public key.

The data member `vector< vector<long> > keySwitchMap` encodes all these graphs G_i in a way that makes it easy to find the sequence of operation needed to implement any given automorphism. For every i, `keySwitchMap[i]` is a vector of indexes that stores information about the graph G_i. Specifically, `keySwitchMap[i][t]` is an index into the vector of key-switching matrices, pointing out the first step in the shortest path $t \rightsquigarrow 1$ in G_i (if any). In other words, if 1 is reachable from t in G_i, then `keySwitchMap[i][t]` is an index k such that `keySwitching`$[k] = W[\mathfrak{s}_i(X^{ts^{-1}}) \Rightarrow \mathfrak{s}_i(X)]$ where s is one step closer to 1 in G_i than t. In particular, if we have in the public key a matrix $W[\mathfrak{s}_i(X^t) \Rightarrow \mathfrak{s}_i(X)]$ then `keySwitchMap[i][t]` contains the index of that matrix. If 1 is not reachable from t in G_i, then `keySwitchMap`[i][t] $= -1$.

The maps in keySwitchMap are built using a breadth-first search on the graph G_i, by calling the method `void setKeySwitchMap(long keyId=0);` This method should be called after all the key-switching matrices are added to the public key. If more matrices are generated later, then it should be called again. Once keySwitchMap is initialized, it is used by the method `Ctxt::smartAutomorph` as follows: to implement $X \mapsto X^t$ on a canonical ciphertext relative to secret key $\mathfrak{s}_i$, we do the following:

```
1. while $t \neq 1$
2.     set $j$ = pubKey.keySwitchMap[i][t]          // matrix index
3.     set matrix = pubKey.keySwitch[j]             // the matrix itself
4.     set $k$ = matrix.fromKey.getPowerOfX()       // the next step
5.     perform automorphism $X \mapsto X^k$, then re-linearize
6.     $t = t \cdot k^{-1} \bmod m$                 // Now we are one step closer to 1
```

The operations in steps 2,3 above are combined in the method

`const KeySwitch& FHEPubKey::getNextKSWmatrix(long t, long i);`

That is, on input t, i it returns the matrix whose index in the list is $j = keySwitchMap[i][t]$. Also, the convenience method `bool FHEPubKey::isReachable(long t, long keyID=0) const` check if `keySwitchMap[keyID][t]` is defined, or it is -1 (meaning that 1 is not reachable from t in the graph $G_{\texttt{keyID}}$).

3.2.3 The FHESecKey class

The FHESecKey class is derived from FHEPubKey, and contains an additional data member with the secret key(s), `vector<DoubleCRT> sKeys`. It also provides methods for key-generation, decryption, and generation of key-switching matrices, as described next.

Key-generation. The FHESecKey class provides methods for either generating a new secret-key polynomial with a specified Hamming weight, or importing a new secret key that was generated by the calling application. That is, we have the methods:

```
long ImportSecKey(const DoubleCRT& sKey, long hwt, long ptxtSpace=0);
long GenSecKey(long hwt, long ptxtSpace=0);
```

For both these methods, if the plaintext-space modulus is unspecified then it is taken to be the default 2^r from the context. The first of these methods takes a secret key that was generated by the application and insert it into the list of secret keys, keeping track of the Hamming weight of the key and the plaintext space modulus which is supposed to be used with this key. The second method chooses a random secret key polynomial with coefficients $-1/0/1$ where exactly hwt of them are non-zero, then it calls ImportSecKey to insert the newly generated key into the list. Both of these methods return the key-ID, i.e., index of the new secret key in the list of secret keys. Also, with every new secret-key polynomial $\mathfrak{s}_i$, we generate and store also a key-switching matrix $W[\mathfrak{s}_i^2(X) \Rightarrow \mathfrak{s}_i(X)]$ for re-linearizing ciphertexts after multiplication.

The first time that ImportSecKey is called for a specific instance, it also generates a public encryption key relative to this first secret key. Namely, for the first secret key $\mathfrak{s}$ it chooses at random a polynomial $\mathfrak{c}_1^* \in_R \mathbb{A}_{Q_{\mathsf{ct}}}$ (where Q_{ct} is the product of all the ciphertext primes) as well as a low-norm error polynomial $\mathfrak{e}^* \in \mathbb{A}_{Q_{\mathsf{ct}}}$ (with Gaussian coefficients), then sets $\mathfrak{c}_0^* := [\texttt{ptxtSpace} \cdot \mathfrak{e}^* - \mathfrak{s} \cdot \mathfrak{c}_1^*]_{Q_{\mathsf{ct}}}$. Clearly the resulting pair $(\mathfrak{c}_0^*, \mathfrak{c}_1^*)$ satisfies $\mathfrak{m}^* \overset{\text{def}}{=} [\mathfrak{c}_0^* + \mathfrak{s} \cdot \mathfrak{c}_1^*]_{Q_{\mathsf{ct}}} = \texttt{ptxtSpace} \cdot \mathfrak{e}^*$, and the noise estimate for this public encryption key is $\texttt{noiseVar}^* = \mathsf{E}[|\mathfrak{m}^*(\tau_m)|^2] = p^2\sigma^2 \cdot \phi(m)$.

Decryption. The decryption process is rather straightforward. We go over all the ciphertext parts in the given ciphertext, multiply each part by the secret key that this part points to, and sum the result modulo the current BGV modulus. Then we reduce the result modulo the plaintext-space modulus, which gives us the plaintext. This is implemented in the method

```
   void Decrypt(ZZX& plaintxt, const Ctxt &ciphertxt) const;
```

that returns the result in the `plaintxt` argument. For debugging purposes, we also provide the method `void Decrypt(ZZX& plaintxt, const Ctxt &ciphertxt, ZZX& f) const`, that returns also the polynomial before reduction modulo the plaintext space modulus. We stress that *it would be insecure to use this method in a production system*, it is provided only for testing and debugging purposes.

Generating key-switching matrices. We also provide an interface for generating key-switching matrices, using the method:

```
   void GenKeySWmatrix(long fromSPower, long fromXPower, long fromKeyIdx=0,
                       long toKeyIdx=0, long ptxtSpace=0);
```

This method checks if the relevant key-switching matrix already exists, and if not then it generates it (as described in Section 3.2.1) and inserts into the list `keySwitching`. If left unspecified, the plaintext space defaults to 2^r, as defined by `context.mod2r`.

Secret-key encryption. We also provide a secret-key encryption method, that produces ciphertexts with a slightly smaller noise than the public-key encryption method. Namely we have the method

```
   long FHESecKey::Encrypt(Ctxt &c, const ZZX& ptxt, long ptxtSpace, long skIdx)
const;
```

that encrypts the polynomial `ptxt` relative to plaintext-space modulus `ptxtSpace`, and the secret key whose index is `skIdx`. Similarly to the choice of the public encryption key, the `Encrypt` method chooses at random a polynomial $\mathfrak{c}_1 \in_R \mathbb{A}_{Q_{\mathsf{ct}}}$ (where Q_{ct} is the product of all the ciphertext primes) as well as a low-norm error polynomial $\mathfrak{e} \in \mathbb{A}_{Q_{\mathsf{ct}}}$ (with Gaussian coefficients), then sets $\mathfrak{c}_0 := [\mathtt{ptxtSpace} \cdot \mathfrak{e} + \mathtt{ptxt} - \mathfrak{s} \cdot \mathfrak{c}_1]_{Q_{\mathsf{ct}}}$. Clearly the resulting pair $(\mathfrak{c}_0, \mathfrak{c}_1)$ satisfies $\mathfrak{m} \overset{\text{def}}{=} [\mathfrak{c}_0 + \mathfrak{s} \cdot \mathfrak{c}_1]_{Q_{\mathsf{ct}}} = \mathtt{ptxtSpace} \cdot \mathfrak{e} + \mathtt{ptxt}$, and the noise estimate for this public encryption key is $\mathtt{noiseVar} \approx \mathsf{E}[|\mathfrak{m}(\tau_m)|^2] = p^2\sigma^2 \cdot \phi(m)$.

3.3 The KeySwitching module: What matrices to generate

This module implements a few useful strategies for deciding what key-switching matrices for automorphism to choose during key-generation. Specifically we have the following methods:

`void addAllMatrices(FHESecKey& sKey, long keyID=0);`
For $i =$ `keyID`, generate key-switching matrices $W[\mathfrak{s}_i(X^t) \Rightarrow \mathfrak{s}_i(X)]$ for all $t \in \mathbb{Z}_m^*$.

`void add1DMatrices(FHESecKey& sKey, long keyID=0);`
For $i =$ `keyID`, generate key-switching matrices $W[\mathfrak{s}_i(X^{g^e}) \Rightarrow \mathfrak{s}_i(X)]$ for every generator g of $\mathbb{Z}_m^* / \langle p \rangle$ with order $\mathsf{ord}(g)$, and every exponent $0 < e < \mathsf{ord}(g)$. Also if the order of g in $\mathbb{Z}_m^*$ is

not the same as its order in $\mathbb{Z}_m^*/\langle p, \ldots \rangle$, then generate also the matrices $W[\mathfrak{s}_i(X^{g^{-e}}) \Rightarrow \mathfrak{s}_i(X)]$ (cf. Section 2.4).

We note that since every $t \in \mathbb{Z}_m^*$ can be expressed as a product $t = \prod_i g_i^{e_i} \bmod m$, then these matrices suffice to implement every automorphism $X \mapsto X^t$. In particular they are enough to implement all the automorphisms that are needed for the data-movement routines from Section 4.

`void addSome1DMatrices(FHESecKey& sKey, long bound=100,long keyID=0);`

For $i =$ `keyID`, we generate just a subset of the matrices that are generated by `add1DMatrices`, so that each of the automorphisms $X \mapsto X^{g^e}$ can be implemented by at most two steps (and similarly for $X \mapsto X^{g^{-e}}$ for generators whose orders in $\mathbb{Z}_m^*$ and $\mathbb{Z}_m^*/\langle p, \ldots \rangle$ are different). In other words, we ensure that the graph G_i (cf. Section 3.2.2) has a path of length at most 2 from g^e to 1 (and also from g^{-e} to 1 for g's of different orders).

In more detail, if $\mathsf{ord}(g) \leq$ `bound` then we generate all the matrices $W[\mathfrak{s}_i(X^{g^e}) \Rightarrow \mathfrak{s}_i(X)]$ (or $W[\mathfrak{s}_i(X^{g^{-e}}) \Rightarrow \mathfrak{s}_i(X)]$) just like in `add1DMatrices`. When $\mathsf{ord}(g) >$ `bound`, however, we generate only $O(\sqrt{\mathsf{ord}(g)})$ matrices for this generator: Denoting $B_g = \lceil \sqrt{\mathsf{ord}(g)} \rceil$, for every $0 < e < B_g$ let $e' = e \cdot B_g \bmod m$, then we generate the matrices $W[\mathfrak{s}_i(X^{g^e}) \Rightarrow \mathfrak{s}_i(X)]$ and $W[\mathfrak{s}_i(X^{g^{e'}}) \Rightarrow \mathfrak{s}_i(X)]$. In addition, if g has a different order in $\mathbb{Z}_m^*$ and $\mathbb{Z}_m^*/\langle p, \ldots \rangle$ then we generate also $W\left[\mathfrak{s}_i(X^{g^{-e'}}) \Rightarrow \mathfrak{s}_i(X)\right]$.

`void addFrbMatrices(FHESecKey& sKey, long keyID=0);`

For $i =$ `keyID`, generate key-switching matrices $W[\mathfrak{s}_i(X^{p^e}) \Rightarrow \mathfrak{s}_i(X)]$ for $0 < e < d$ where d is the order of p in $\mathbb{Z}_m^*$. (Recall that p is the plaintext-space prime.)

4 The Data-Movement Layer

At the top level of our library, we provide some interfaces that allow the application to manipulate arrays of plaintext values homomorphically. The arrays are translated to plaintext polynomials using the encoding/decoding routines provided by PAlgebraMod (cf. Section 2.5), and then encrypted and manipulated homomorphically using the lower-level interfaces from the crypto layer.

4.1 The class **EncryptedArray**

This class presents the plaintext values to the application as either a linear array (with as many entries as there are elements in $\mathbb{Z}_m^*/\langle p \rangle$), or as a multi-dimensional array corresponding to the structure of the group $\mathbb{Z}_m^*/\langle p \rangle$.

The definition of this class has a similar structure to that of `PAlgebraMod`:

- `EncryptedArrayBase` is a virtual base class
- `EncryptedArrayDerived<type>` is a derived template class, where `type` is either `PA_GF2` or `PA_zz_p`.
- The class `EncryptedArray` itself is a simple wrapper around a smart pointer to a `EncryptedArrayBase` object: copying a `EncryptedArray` object results is a "deep copy" of the underlying object of the derived class.

For EncryptedArray we have the constructor

```
    EncryptedArray(const FHEcontext& context, const ZZX& G = ZZX(1, 1));
```

taking as input the context (that specifies m, p, and r, among other things), and a polynomial G that defines the slot subring $\mathbb{Z}_{p^r}[X]/(G)$. The default value for the polynomial is $G(X) = X$, resulting in plaintext values in the base ring $\mathbb{Z}_{p^r}$.

The multi-dimensional array view. This view arranges the plaintext slots in a multi-dimensional array, corresponding to the structure of $\mathbb{Z}_m^*/\langle p\rangle$. The number of dimensions is the number of generators of $\mathbb{Z}_m^*/\langle p\rangle$, and the size along the i'th dimension is the order of the i'th generator.

Recall from Section 2.4 that each plaintext slot is represented by some $t \in \mathbb{Z}_m^*$, such that the set of representatives $T \subset \mathbb{Z}_m^*$ has exactly one element from each conjugacy class of $\mathbb{Z}_m^*/\langle p\rangle$. Moreover, if $f_1, \ldots, f_n \in T$ are the generators of $\mathbb{Z}_m^*/\langle p\rangle$ (with f_i having order $\mathsf{ord}(f_i)$), then every $t \in T$ can be written uniquely as $t = [\prod_i f_i^{e_i}]_m$ with each exponent e_i taken from the range $0 \le e_i < \mathsf{ord}(f_i)$. The generators are roughly arranged by their order (i.e., $\mathsf{ord}(f_i) \ge \mathsf{ord}(f_{i+1})$), except that we put all the generators g_i that have the same order in $\mathbb{Z}_m^*$ and $\mathbb{Z}_m^*/\langle p, g_1, \ldots, g_{i-1}\rangle$ before all the other generators.

Hence the multi-dimensional-array view of the plaintext slots will have them arranged in a n-dimensional hypercube, with the size of the i'th side being $\mathsf{ord}(f_i)$. Every entry in this hypercube is indexed by some $\vec{e} = (e_1, e_2, \ldots, e_n)$, and it contains the plaintext slot associated with the representative $t = [\prod_i f_i^{e_i}]_m \in T$. (Note that the lexicographic order on the vectors $\vec{e}$ of indexes induces a linear ordering on the plaintext slots, which is what we use in our linear-array view described below.) The multi-dimensional-array view provides the following interfaces:

`long dimension() const;` returns the dimensionality (i.e., the number of generators in $\mathbb{Z}_m^*/\langle p\rangle$).

`long sizeOfDimension(long i);` returns the size along the i'th dimension (i.e., $\mathsf{ord}(f_i)$).

`long coordinate(long i, long k) const;` return the i'th entry of the k'th vector in lexicographic order.

`void rotate1D(Ctxt& ctxt, long i, long k) const;`

This method rotates the hypercube by k positions along the i'th dimension, moving the content of the slot indexed by $(e_1 \ldots, e_i, \ldots e_n)$ to the slot indexed by $(e_1 \ldots, e_i + k, \ldots e_n)$, addition modulo $\mathsf{ord}(f_i)$. Note that the argument k above can be either positive or negative, and rotating by $-k$ is the same as rotating by $\mathsf{ord}(f_i) - k$.

The `rotate1D` operation is closely related to the "native" automorphism operation of the lower-level `Ctxt` class. Indeed, if f_i has the same order in $\mathbb{Z}_m^*$ as in $\mathbb{Z}_m^*/\langle p, \ldots\rangle$ then we just apply the automorphism $X \mapsto X^{f_i^k}$ on the input ciphertext using `ctxt.smartAutomorph`(f_i^k). If f_i has different orders in $\mathbb{Z}_m^*$ and $\mathbb{Z}_m^*/\langle p, \ldots\rangle$ then we need to apply the two automorphisms $X \mapsto X^{f_i^k}$ and $X \mapsto X^{f_i^{k-\mathsf{ord}(f_i)}}$ and then "mix and match" the two resulting ciphertexts to pick from each of them the plaintext slots that did not undergo wraparound (see description of the `select` method below).

`void shift1D(Ctxt& ctxt, long i, long k) const;`

This is similar to `rotate1D`, except it implements a non-cyclic shift with zero fill. Namely, for a positive $k > 0$, the content of any slot indexed by $(e_1 \ldots, e_i, \ldots e_n)$ with $e_i < \mathsf{ord}(f_i) - k$ is moved to the slot indexed by $(e_1 \ldots, e_i + k, \ldots e_n)$, and all the other slots are filled with zeros. For a negative $k < 0$, the content of any slot indexed by $(e_1 \ldots, e_i, \ldots e_n)$ with $e_i \geq |k|$ is moved to the slot indexed by $(e_1 \ldots, e_i + k, \ldots e_n)$, and all the other slots are filled with zeros.

The operation is implemented by applying the corresponding automorphism(s), and then zero-ing out the wraparound slots by multiplying the result by a constant polynomial that has zero in these slots.

The linear array view. This view arranges the plaintext slots in a linear array, with as many entries as there are plaintext slots (i.e., $|\mathbb{Z}_m^* / \langle p \rangle|$). These entries are ordered according to the lexicographic order on the vectors of indexes from the multi-dimensional array view above. In other words, we obtain a linear array simply by "opening up" the hypercube from above in lexicographic order. The linear-array view provides the following interfaces:

`long size() const;` returns the number of entries in the array, i.e., the number of plaintext slots.

`void rotate(Ctxt& ctxt, long k) const;`

Cyclically rotate the linear array by k positions, moving the content of the j'th slot (by the lexicographic order) to slot $j + k$, addition modulo the number of slots. (Below we denote the number of slots by N.) Rotation by a negative number $-N < k < 0$ is the same as rotation by the positive amount $k + N$.

The procedure for implementing this cyclic rotation is roughly a concurrent version of the grade-school addition-with-carry procedure, building on the multidimensional rotations from above. What we need to do is to add k (modulo N) to the index of each plaintext slot, all in parallel. To that end, we think of the indexes (and the rotation amount k) as they are represented in the lexicographic order above. Namely, we identify k with the k'th vector in the lexicographic order, denoted $\vec{e}^{(k)} = (e_1^{(k)}, \ldots, e_n^{(k)})$. (Similarly, we identify each index j with the j'th vector in that order). We can now think of rotation by k as adding the multi-precision vector $\vec{e}^{(k)}$ to all the vectors $\vec{e}^{(j)}$, $j = 0, 1, \ldots, N - 1$ in parallel.

Beginning with the least-significant digit in these vector, we use rotate-by-$e_n^{(k)}$ along the n'th dimension to implement the operation of $e_n^{(j)} = e_n^{(j)} + e_n^{(k)} \bmod \mathsf{ord}(f_n)$ for all j at once.

Moving to the next digit, we now have to add to each $e_{n-1}^{(j)}$ either $e_{n-1}^{(k)}$ or $1 + e_{n-1}^{(k)}$, depending on whether or not there was a carry from the previous position. To do that, we compute two rotation amount along the $(n-1)$'th dimension, by $e_{n-1}^{(k)}$ and $1 + e_{n-1}^{(k)}$, then use a MUX operation to choose the right rotation amount for every slot. Namely, indexes j for which $e_n^{(j)} \geq \mathsf{ord}(f_i) - e_n^{(k)}$ (so we have a carry) are taken from the copy that was rotated by $1 + e_{n-1}^{(k)}$, while other indexes j are taken from the copy that was rotated by $e_{n-1}^{(k)}$.

The MUX operation is implemented by preparing a constant polynomial that has 1's in the slots corresponding to indexes $(e_1, \ldots, e_n)$ with $e_n \geq \mathsf{ord}(f_i) - e_n^{(k)}$ and 0's in all the other slots (call this polynomial `mask`), then computing $\vec{c}' = \vec{c}_1 \cdot \mathtt{mask} + \vec{c}_2 \cdot (1 - \mathtt{mask})$, where $\vec{c}_1, \vec{c}_2$

are the two ciphertexts generated by rotation along dimension $n-1$ by $1+e^{(k)}_{n-1}$ and $e^{(k)}_{n-1}$, respectively.

We then move to the next digit, preparing a mask for those j's for which we have a carry into that position, then rotating by $1+e^{(k)}_{n-2}$ and $e^{(k)}_{n-2}$ along the $(n-2)$'nd dimension and using the mask to do the MUX between these two ciphertexts. We proceed in a similar manner until the most significant digit. To complete the description of the algorithm, note that the mask for processing the i'th digit is computed as follows: For each index j, which is represented by the vector $(e^{(j)}_1 \ldots, e^{(j)}_i, \ldots e^{(j)}_n)$, we have $\texttt{mask}_i[j] = 1$ if either $e^{(j)}_i \geq \mathsf{ord}(f_i) - e^{(k)}_i$, or if $e^{(j)}_i = \mathsf{ord}(f_i) - e^{(k)}_i - 1$ and $\texttt{mask}_{i-1}[j] = 1$ (i.e. we had a carry from position $i-1$ to position i). Hence the rotation procedure works as follows:

```
Rotate($\vec{c}, k$):
  0. Let $(e^{(k)}_1, \ldots, e^{(k)}_n)$ be the $k$'th vector in lexicographic order.
  1. $M_n$ := all-1 mask                    // $M_n$ is a polynomial with 1 in all the slots
  2. Rotate $\vec{c}$ by $e^{(k)}_n$ along the $n$'th dimension
  3. For $i = n-1$ down to 1
  4.     $M'_i$ := 1 in the slots $j$ with $e^{(j)}_{i+1} \geq \mathsf{ord}(f_{i+1}) - e^{(k)}_{i+1}$,  0 in all the other slots
  5.     $M''_i$ := 1 in the slots $j$ with $e^{(j)}_{i+1} = \mathsf{ord}(f_{i+1}) - e^{(k)}_{i+1} - 1$,  0 in all the outer slots
  6.     $M_i := M'_i + M''_i \cdot M_{i+1}$          // The $i$'th mask
  7.     $\vec{c}\,'$ := rotate $\vec{c}$ by $e^{(k)}_i$ along the $i$'th dimension
  8.     $\vec{c}\,''$ := rotate $\vec{c}$ by $1+e^{(k)}_i$ along the $i$'th dimension
  9.     $\vec{c} := \vec{c}\,'' \cdot M_i + \vec{c}\,' \cdot (1 - M_i)$
 10. Return $\vec{c}$.
```

Note that the mask polynomials are precomputed and stored in the corresponding `PAlgebraMod` object (as they are independent of G and hence may be used for different G's).

`void shift(Ctxt& ctxt, long k) const;` Non-cyclic shift of the linear array by k positions, with zero-fill. For a positive $k > 0$, then every slot $j \geq k$ gets the content of slot $j-k$, and every slot $j < k$ gets zero. For a negative $k < 0$, every slot $j < N - |k|$ gets the content of slot $j + |k|$, and every slot $j \geq N - |k|$ gets zero (with N the number of slots).

For $k > 0$, this procedure is implemented very similarly to the `rotate` procedure above, except that in the last iteration (processing the most-significant digit) we replace the operation of rotate-by-$e^{(k)}_1$ along the 1'st dimension by shift-by-$e^{(k)}_1$ along the 1'st dimension (and similarly use shift-by-$(1+e^{(k)}_1)$ rather than rotate-by-$(1+e^{(k)}_1)$). For a negative amount $-N < k < 0$, we use the same procedure upto the last iteration with amount $N+k$, and in the last iteration use shift-by-e' and shift-by-$(1+e')$ along the 1st dimension, for the negative number $e' = e^{(k)}_1 - \mathsf{ord}(f_i)$.

Other operations. In addition to the following rotation methods, the class EncryptedArray also provides convenience methods that handle both encoding and homomorphic operations in one shot.

Some of the methods that are provided are described below. The class `PlaintextArray` is a convenience class discussed further below.

```
  // Encode the given array in a polynomial
  void encode(ZZX& ptxt, const vector<long>& array) const;
  void encode(ZZX& ptxt, const vector<ZZX>& array) const;
  void encode(ZZX& ptxt, const PlaintextArray& array) const;

  // Decode the given polynomial into an array of plaintext values
  void decode(vector< long  >& array, const ZZX& ptxt) const;
  void decode(vector< ZZX  >& array, const ZZX& ptxt) const;
  void decode(PlaintextArray& array, const ZZX& ptxt) const;

  // Encode the array in a polynomial, then encrypt it in the ciphertext c
  void encrypt(Ctxt& ctxt, const FHEPubKey& pKey,
               const vector< long >& ptxt) const;
  void encrypt(Ctxt& ctxt, const FHEPubKey& pKey,
               const vector< ZZX >& ptxt) const;
  void encrypt(Ctxt& ctxt, const FHEPubKey& pKey,
               const PlaintextArray& ptxt) const;

  // Decrypt the ciphertext c, then decode the result into the array
  void decrypt(const Ctxt& ctxt, const FHESecKey& sKey,
               vector< long >& ptxt) const;
  void decrypt(const Ctxt& ctxt, const FHESecKey& sKey,
               vector< ZZX >& ptxt) const;
  void decrypt(const Ctxt& ctxt, const FHESecKey& sKey,
               PlaintextArray& ptxt) const;

  // compute coefficients of a linear polynomial of a given linear map
  void buildLinPolyCoeffs(vector<ZZX>& C, const vector<ZZX>& L) const;

  // MUX: for p=encode(selector), set c1 = c1*p + c2*(1-p)
  void select(Ctxt& ctxt1, const Ctxt& ctxt2,
              const vector< long >& selector) const;
  void select(Ctxt& ctxt1, const Ctxt& ctxt2,
              const vector< ZZX >& selector) const;
  void select(Ctxt& ctxt1, const Ctxt& ctxt2,
              const PlaintextArray& selector) const;
```

4.2 The class `PlaintextArray`

This is a convenience class provided to simplify testing. It has a structure that mirrors that of `PAlgebraMod` and `EncryptedArray`. Essentially, an object of type `PlaintextArray` is a vector of polynomials over $\mathbb{Z}_{p^r}[X]$ modulo G, and is associated with a particular `EncryptedArray` object (which, among other things, defined p, r, and G). These polynomials are represented as objects of type `GF2X` or `zz_pX`, depending on the values p and r, in a way that is compatible with the representation used in the corresponding `PAlgebraMod` and `EncryptedArray` objects. The length of a `PlaintextArray` vector is equal to the number of slots.

A `PlaintextArray` object is created using a constructor that specifies the corresponding `EncryptedArray` object, which remains bound to the constructed `PlaintextArray` object throughout its lifetime. The following methods are provided (bear in mind that all arithmetic is polynomial arithmetic modulo p^r and G):

```
// rotate and shift
void rotate(long k);
void shift(long k);

// encoding
void encode(const vector< long >& array);
void encode(const vector< ZZX >& array);

// decoding
void decode(vector< long  >& array);
void decode(vector< ZZX  >& array)'

// encode the same value in each slot
void encode(long val) { rep->encode(val); }
void encode(const ZZX& val) { rep->encode(val); }

// replicate the value in slot i in all other slots
void replicate(long i);

// generate a random array
void random();

// equality testing
bool equals(const PlaintextArray& other) const;
bool equals(const vector<long>& other) const;
bool equals(const vector<ZZX>& other) const;


// slot-wise arithmetic
void add(const PlaintextArray& other);
void sub(const PlaintextArray& other);
void mul(const PlaintextArray& other);
void negate();
```

5 Using the Library

The following code illustrates how the library can be used to homomorphically evaluate a simple circuit. This code depends on several parameters:

- m, p, r — the native plaintext space is $\mathbb{Z}[X]/(\Phi_m(X), p^r)$,

- L — the number "levels," i.e. the number of ciphertext primes (cf. Section 2.7),
- c — number of columns in key switching matrix (recommended $c = 2$ or $c = 3$),
- w — the Hamming weight of a secret key ($w = 64$ recommended),
- G (of type ZZX) — a monic polynomial, irreducible over $\mathbb{Z}_p$.

  If $r = 1$, we only require that the degree of G divides the degree of the irreducible factors of $\Phi_m(X)$ modulo p.

  If $r > 1$, G must either have degree 1, or be equal to the first factor of $\Phi_m(X)$ modulo p^r (as computed by the library). That factor is accessible in the code below as context.alMod.getFactorsOverZZ()[0].

```
/*************** Basic usage of the HElib library ***************/
{ long m, p, r, L, c, w; // parameters
  ZZX G;                 // defines the plaintext space

  // Some code here to choose all the parameters, perhaps
  // using the fucntion FindM(...) in the FHEContext module

  FHEcontext context(m, p, r);
  // initialize context

  buildModChain(context, L, c);
  // modify the context, adding primes to the modulus chain

  FHESecKey secretKey(context);
  // construct a secret key structure associated with the context

  const FHEPubKey& publicKey = secretKey;
  // an "upcast": FHESecKey is a subclass of FHEPubKey

  secretKey.GenSecKey(w);
  // actually generate a secret key with Hamming weight w

  addSome1DMatrices(secretKey);
 // compute key-switching matrices that we need

  EncryptedArray ea(context, G);
  // constuct an Encrypted array object ea that is
  // associated with the given context and the polynomial G

  long nslots = ea.size();
  // number of plaintext slots

  PlaintextArray p0(ea);
  PlaintextArray p1(ea);
  PlaintextArray p2(ea);
  PlaintextArray p3(ea);
  // PlaintextArray objects associated with the given EncryptedArray ea
```

```
p0.random();
p1.random();
p2.random();
p3.random();
// generate random plaintexts: slots initalized with random elements of Z[X]/(G,p^r)

Ctxt c0(publicKey), c1(publicKey), c2(publicKey), c3(publicKey);
// construct ciphertexts associated with the given public key

ea.encrypt(c0, publicKey, p0);
ea.encrypt(c1, publicKey, p1);
ea.encrypt(c2, publicKey, p2);
ea.encrypt(c3, publicKey, p3);
// encrypt each PlaintextArray

long shamt = RandomBnd(2*(nslots/2) + 1) - (nslots/2);
     // shift-amount: random number in [-nslots/2..nslots/2]

long rotamt = RandomBnd(2*nslots - 1) - (nslots - 1);
     // rotation-amount: random number in [-(nslots-1)..nslots-1]

PlaintextArray const1(ea);
PlaintextArray const2(ea);
const1.random();
const2.random();
// two random constants

// Perform some simple computations directly on the plaintext arrays:

p1.mul(p0); // p1 = p1 * p0 (slot-wise modulo G)
p0.add(const1); // p0 = p0 + const1
p2.mul(const2); // p2 = p2 * const2
PlaintextArray tmp_p(p1); // tmp = p1
tmp_p.shift(shamt); // shift tmp_p by shamt
p2.add(tmp_p); // p2 = p2 + tmp_p
p2.rotate(rotamt); // rotate p2 by rotamt
p1.negate(); // p1 = - p1
p3.mul(p2); // p3 = p3 * p2
p0.sub(p3); // p0 = p0 - p3

// Perform the same operations on the ciphertexts

ZZX const1_poly, const2_poly;
ea.encode(const1_poly, const1);
ea.encode(const2_poly, const2);
// encode const1 and const2 as plaintext polynomials

c1.multiplyBy(c0); // c1 = c1 * c0
c0.addConstant(const1_poly); // c0 = c0 + const1
c2.multByConstant(const2_poly); // c2 = c2 * const2
```

```
  Ctxt tmp(c1); // tmp = c1
  ea.shift(tmp, shamt); // shift tmp by shamt
  c2 += tmp; // c2 = c2 + tmp
  ea.rotate(c2, rotamt); // rotate c2 by shamt
  c1.negate(); // c1 = - c1
  c3.multiplyBy(c2); // c3 = c3 * c2
  c0 -= c3; // c0 = c0 - c3

  // Decrypt the ciphertexts and compare

  PlaintextArray pp0(ea);
  PlaintextArray pp1(ea);
  PlaintextArray pp2(ea);
  PlaintextArray pp3(ea);

  ea.decrypt(c0, secretKey, pp0);
  ea.decrypt(c1, secretKey, pp1);
  ea.decrypt(c2, secretKey, pp2);
  ea.decrypt(c3, secretKey, pp3);

  if (!pp0.equals(p0)) cerr << "oops 0\n";
  if (!pp1.equals(p1)) cerr << "oops 1\n";
  if (!pp2.equals(p2)) cerr << "oops 2\n";
  if (!pp3.equals(p3)) cerr << "oops 3\n";
}
```

5.1 Encrypted Arrays

The code fragment in Figure 2 illustrates some of the finer points of using the classes `PAlgebraMod`, `EncryptedArray`, and `PlaintextArray`. These classes share the same design structure: the class itself is implemented as a smart pointer to a virtual class, and for the latter, concrete subclasses are used which depend on the underlying data types used to represent polynomials and related types.

In line 8, `context.alMod` is an object of type `PAlgebraMod`, whose method `getTag()` returns a value of the enumeration type `PA_tag`, which is either `PA_GF2_tag` or `PA_zz_p_tag`.

Lines 11 and 17 show how to dynamically "downcast" a `PAlgebraMod` object to a corresponding type that is specialized to either `GF2` or `zz_p`. With such a specialized object, one access some lower level methods that are not directly available to a `PAlgebraMod` object; in particular, one can access methods that directly receive or return references to objects of underlying polynomial type, `GF2X` or `zz_pX`, rather than working with more generic polynomials of type `ZZX`. Lines 13 and 20 illustrate this by calling the method `getFactors()`, which returns a reference to a vector of polynomials of a type appropriate to the underlying polynomial type.

Line 19 illustrates that before accessing or using any polynomials represented by the `zz_pX` type, the "global modulus context" for the `zz_p` type must be restored (this is an artifact of how NTL implements such moduli). In NTL, the code `zz_pBak bak; bak.save();` will save a "backup" of the current modulus associated with `zz_p` in the local variable `bak`. After that, the code `alMod.restoreContext();` will restore the modulus associated with the plaintext space (which is p^r). When the object `bak` is destroyed at the end of its enclosing scope, as a side effect of the corresponding destructor, the original modulus associated with `zz_p` will be restored.

Although there is no type corresponding to `zz_pBak` for `GF2`, the class `PA_GF2` provides `typedef`'s

```
long m, p, r;

FHEcontext context(m, p, r);

FHESecKey secretKey(context);
const FHEPubKey& publicKey = secretKey;

switch (context.alMod.getTag()) {

   case PA_GF2_tag: {
      const PAlgebraModDerived<PA_GF2>& alMod =
         context.alMod.getDerived(PA_GF2());
      const vec_GF2X& factors = alMod.getFactors();
   }

   case PA_zz_p_tag: {
      const PAlgebraModDerived<PA_zz_p>& alMod =
         context.alMod.getDerived(PA_zz_p());
      zz_pBak bak; bak.save(); alMod.restoreContext();
      const vec_zz_pX& factors = alMod.getFactors();
   }
}
```

Figure 2: Using the `EncryptedArray` classes

that help facilitate template programming. Access to the specializations for `EncryptedArray` and `PlaintextArray` is achieved in a similar manner. For more details, please see the documentation in the header files `PAlgebraMod.h` and `EncryptedArray.h`.

References

[1] L. I. Bluestein. A linear filtering approach to the computation of the discrete fourier transform. Northeast Electronics Research and Engineering Meeting Record 10, 1968.

[2] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.

[3] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009.

[4] C. Gentry, S. Halevi, and N. Smart. Fully homomorphic encryption with polylog overhead. In *"Advances in Cryptology - EUROCRYPT 2012"*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.

[5] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *"Advances in Cryptology - CRYPTO 2012"*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012. Full version at `http://eprint.iacr.org/2012/099`.

[6] C. Gentry, S. Halevi, and N. P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography - PKC 2012*, volume 7293 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[7] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *Advances in Cryptology - EUROCRYPT'10*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[8] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–177. Academic Press, 1978.

[9] V. Shoup. NTL: A Library for doing Number Theory. `http://shoup.net/ntl/`, Version 5.5.2, 2010.

[10] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.

A Proof of noise-estimate

We observed empirically that for a random Hamming-weight-H polynomial $\mathfrak{s}$ with coefficients $-1/0/1$ we have $\mathsf{E}[|\mathfrak{s}^r(\tau)|^{2r}] \approx r! \cdot H^r$, where r is an integer and τ is the principal complex m-th root of unity, $\tau = e^{2\pi i/m}$. Below we prove that this is indeed a good approximation when r is small enough relative to H, m. To simplify the proof, we analyze the case that each coefficient of $\mathfrak{s}$ is chosen uniformly at random from $-1/0/1$, so that the expected Hamming weight is H. Also, we assume that $\mathfrak{s}$ is chosen as a degree-$(m-1)$ polynomial (rather than degree $\phi(m)-1$).

Theorem 1. *Suppose m, r, H are positive integers, with $H \le m$, and let $\tau = e^{2\pi i/m} \in \mathbb{C}$. Suppose that we choose $f_0, \ldots, f_{m-1}$ independently, where for $i = 0, \ldots, m-1$, f_i is ± 1 with probability $H/2m$ each and 0 with probability $1 - H/m$. Let $f(X) = \sum_{i=0}^{m-1} f_i X^i$. Then for fixed r and $H, m \to \infty$, we have*

$$\mathsf{E}[|f(\tau)^r|^2] = \mathsf{E}[|f(\tau)^{2r}|] \sim r! H^r.$$

In particular, for $H \ge 2r^2$, we have

$$\left| \frac{\mathsf{E}[|f(\tau)^{2r}|]}{r! H^r} - 1 \right| \le \frac{2r^2}{H} + \frac{2^{r+1} r^2}{m}.$$

Before proving Theorem 1, we introduce some notation and prove some technical results that will be useful. Recall the "falling factorial" notation: for integers n, k with $0 \le k \le n$, we define $n^{\underline{k}} = \prod_{j=0}^{k-1}(n-j)$.

Lemma 1. *For $n \ge k^2 > 0$, we have $n^{\underline{k}} \ge n^k - k^2 n^{k-1}$.*

Proof. We have

$$n^{\underline{k}} \ge (n-k)^k = n^k - \binom{k}{1} k n^{k-1} + \binom{k}{2} k^2 n^{k-2} - \binom{k}{3} k^3 n^{k-3} + - \cdots .$$

The lemma follows by verifying that when $n \ge k^2$, in the above binomial expansion, the sum of every consecutive positive/negative pair of terms in non-negative. □

Lemma 2. *For $n \geq 2k^2 > 0$, we have $n^{\underline{k}} \geq n^k/2$.*

Proof. This follows immediately from the previous lemma. $\square$

Next, we recall the notion of the *Stirling number of the second kind*, which is the number of ways to partition a set of ℓ objects into k non-empty subsets, and is denoted $\left\{ {\ell \atop k} \right\}$. We use the following standard result:

$$\sum_{k=1}^{\ell} \left\{ {\ell \atop k} \right\} n^{\underline{k}} = n^{\ell}. \tag{1}$$

Finally, we define M_{2n} to be the number of perfect matchings in the complete graph on $2n$ vertexes, and $M_{n,n}$ to be the number of perfect matchings on the complete bipartite graph on two sets of n vertexes. The following facts are easy to establish:

$$M_{n,n} = n! \tag{2}$$

and

$$M_{2n} \leq 2^n n!. \tag{3}$$

We now turn to the proof of the theorem. We have

$$|f(\tau)^{2r}| = |f(\tau)^r|^2 = f(\tau)^r f(\bar{\tau})^r = \sum_{i_1,\ldots,i_{2r}} f_{i_1} \cdots f_{i_{2r}} \cdot \tau^{i_1} \cdots \tau^{i_r} \cdot \tau^{-i_{r+1}} \cdots \tau^{-i_{2r}}.$$

We use the usual notion of expected values of complex-valued random variables: if U and V are real-valued random variables, then $\mathsf{E}[U + V\mathbf{i}] = \mathsf{E}[U] + \mathsf{E}[V]\mathbf{i}$. The usual rules for sums and products of expectations work just as for real-valued random variables. By linearity of expectation, we have

$$\mathsf{E}[|f(\tau)^{2r}|] = \sum_{i_1,\ldots,i_{2r}} \mathsf{E}[f_{i_1} \cdots f_{i_{2r}}] \cdot \tau^{i_1} \cdots \tau^{i_r} \cdot \tau^{-i_{r+1}} \cdots \tau^{-i_{2r}}. \tag{4}$$

Here, each index i_t runs over the set $\{0, \ldots, m-1\}$. In this sum, because of independence and the fact that any odd power of f_i has expected value 0, the only terms that contribute a non-zero value are those in which each index value occurs an even number of times, in which case, if there are k distinct values among $i_1, \ldots, i_{2r}$, we have

$$\mathsf{E}[f_{i_1} \cdots f_{i_{2r}}] = (H/m)^k.$$

We want to regroup the terms in (4). To this end, we introduce some notation: for an integer $t \in \{1, \ldots, 2r\}$ define $w(t) = 1$ if $t \leq r$, and $w(t) = -1$ if $t > r$; for a subset $e \subseteq \{1, \ldots, 2r\}$, define $w(e) = \sum_{t \in e} w(t)$. We call $w(e)$ the "weight" of e. Then we have:

$$\mathsf{E}[|f(\tau)^{2r}|] = \sum_{P=\{e_1,\ldots,e_k\}} (H/m)^k \sum_{j_1,\ldots,j_k}{}' \tau^{j_1 w(e_1) + \cdots + j_k w(e_k)}. \tag{5}$$

Here, the outer summation is over all k and all "even" partitions $P = \{e_1, \ldots, e_k\}$ of the set $\{1, \ldots, 2r\}$, where each element of the partition has an even cardinality. The inner summation is over all sequences of indexes $j_1, \ldots, j_k$, where each index runs over the set $\{0, \ldots, m-1\}$, but where no value in the sequence is repeated — the special summation notation $\sum'_{j_1,\ldots,j_k}$ emphasizes this restriction.

Since $|\tau| = 1$, it is clear that

$$\left|\mathsf{E}[|f(\tau)^{2r}|] - \sum_{P=\{e_1,\dots,e_k\}} (H/m)^k \sum_{j_1,\dots,j_k} \tau^{j_1 w(e_1)+\cdots+j_k w(e_k)}\right| \le \sum_{P=\{e_1,\dots,e_k\}} (H/m)^k (m^k - m^{\underline{k}}) \quad (6)$$

Note that in this inequality the inner sum on the left is over *all* sequences of indexes $j_1, \dots, j_k$, without the restriction that the indexes in the sequence are unique.

Our first task is to bound the sum on the right-hand side of (6). Observe that any even partition $P = \{e_1, \dots, e_k\}$ can be formed by merging the edges of some perfect matching on the complete graph on vertexes $\{1, \dots, 2r\}$. So we have

$$\begin{aligned}
\sum_{P=\{e_1,\dots,e_k\}} (H/m)^k (m^k - m^{\underline{k}}) &\le \sum_{P=\{e_1,\dots,e_k\}} (H/m)^k k^2 m^{k-1} \qquad \text{(by Lemma 1)} \\
&\le \frac{r^2}{m} \sum_{P=\{e_1,\dots,e_k\}} H^k \\
&\le \frac{r^2}{m} M_{2r} \sum_{k=1}^{r} \left\{ {r \atop k} \right\} H^k \qquad \text{(partitions formed from matchings)} \\
&\le \frac{r^2 2^r r!}{m} \sum_{k=1}^{r} \left\{ {r \atop k} \right\} H^k \qquad \text{(by (3))} \\
&\le \frac{r^2 2^{r+1} r!}{m} \sum_{k=1}^{r} \left\{ {r \atop k} \right\} H^{\underline{k}} \qquad \text{(by Lemma 2)} \\
&= \frac{r^2 2^{r+1} r!}{m} H^r \qquad \text{(by 1).}
\end{aligned}$$

Combining this with (6), we have

$$\left|\mathsf{E}[|f(\tau)^{2r}|] - \sum_{P=\{e_1,\dots,e_k\}} (H/m)^k \sum_{j_1,\dots,j_k} \tau^{j_1 w(e_1)+\cdots+j_k w(e_k)}\right| \le r! H^r \cdot \frac{2^{r+1} r^2}{m}. \quad (7)$$

So now consider the inner sum in (7). The weights $w(e_1), \dots, w(e_k)$ are integers bounded by r in absolute value, and r is strictly less than m by the assumption $2r^2 \le H \le m$. If any weight, say $w(e_1)$, is non-zero, then $\tau^{w(e_1)}$ has multiplicative order dividing m, but not 1, and so the sum $\sum_j \tau^{j w(e_1)}$ vanishes, and hence

$$\sum_{j_1,\dots,j_k} \tau^{j_1 w(e_1)+\cdots+j_k w(e_k)} = \left(\sum_{j_1} \tau^{j_1 w(e_1)}\right)\left(\sum_{j_2,\dots,j_k} \tau^{j_2 w(e_2)+\cdots+j_k w(e_k)}\right) = 0.$$

Otherwise, if all the weights are $w(e_1), \dots, w(e_k)$ are zero, then

$$\sum_{j_1,\dots,j_k} \tau^{j_1 w(e_1)+\cdots+j_k w(e_k)} = m^k.$$

We therefore have

$$\sum_{P=\{e_1,\dots,e_k\}} (H/m)^k \sum_{j_1,\dots,j_k} \tau^{j_1 w(e_1)+\cdots+j_k w(e_k)} = \sum_{\substack{P=\{e_1,\dots,e_k\} \\ w(e_1)=\cdots=w(e_k)=0}} H^k, \quad (8)$$

Observe that any partition $P = \{e_1, \ldots, e_k\}$ with $w(e_1) = \cdots = w(e_k) = 0$ can be formed by merging the edges of some perfect matching on the complete bipartite graph with vertex sets $\{1, \ldots, r\}$ and $\{r+1, \ldots, 2r\}$. The total number of such matchings is $r!$ (see (2)). So we have

$$\begin{aligned} r!H^r \le \sum_{\substack{P=\{e_1,\ldots,e_k\} \\ w(e_1)=\cdots=w(e_k)=0}} H^k &\le r!H^r + r!\sum_{k=1}^{r-1} \left\{ {r \atop k} \right\} H^k \\ &\le 2r!\sum_{k=1}^{r-1} \left\{ {r \atop k} \right\} H^{\underline{k}} \qquad \text{(by Lemma 2)} \\ &= 2r!(H^r - H^{\underline{r}}) \qquad \text{(by (1))} \\ &\le 2r!r^2H^{r-1} \qquad \text{(by Lemma 1)} \end{aligned}$$

Combining this with (7) and (8) proves the theorem.