HElib

1.0

Generated by Doxygen 1.8.3.1

Mon Apr 15 2013 17:19:32

Contents

Chapter 1

HElib Documentation

HElib is 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. HElib is written in C++ and uses the NTL mathematical library. It is distributed under the terms of the GNU General Public License (GPL).

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 AltCRT Class Reference

A single-CRT representation of a ring element.

```
#include <AltCRT.h>
```

Public Member Functions

- AltCRT (const ZZX &poly, const FHEcontext &_context)

  Initializing AltCRT from a ZZX polynomial.
- AltCRT (const ZZX &poly, const FHEcontext &_context, const IndexSet &indexSet)
- AltCRT (const ZZX &poly)

  Context is not specified, use the "active context".
- AltCRT (const FHEcontext &_context)

  Without specifying a ZZX, we get the zero polynomial.
- AltCRT (const FHEcontext &_context, const IndexSet &indexSet)

  Also specify the IndexSet explicitly.
- AltCRT & **operator=** (const AltCRT &other)
- AltCRT & **operator=** (const SingleCRT &other)
- AltCRT & **operator=** (const ZZX &poly)
- AltCRT & **operator=** (const ZZ &num)
- AltCRT & **operator=** (const long num)
- void toPoly (ZZX &p, bool positive=false) const

  Recovering the polynomial in coefficient representation.
- void toPoly (ZZX &p, const IndexSet &s, bool positive=false) const

  Recovering the polynomial in coefficient representation. This yields an integer polynomial with coefficients in [--P/2,P/2], unless the positive flag is set to true, in which case we get coefficients in [0,P-1] (P is the product of all moduli used). Using the optional IndexSet param we compute the polynomial reduced modulo the product of only the ptimes in that set.
- bool **operator==** (const AltCRT &other) const
- bool **operator!=** (const AltCRT &other) const
- AltCRT & **SetZero** ()
- AltCRT & **SetOne** ()
- void breakIntoDigits (vector$<$ AltCRT $>$ &dgts, long n) const

  Break into n digits,according to the primeSets in context.digits. See Section 3.1.6 of the design document (re-linearization)
- void addPrimes (const IndexSet &s1)

  Expand the index set by s1. It is assumed that s1 is disjoint from the current index set.

- double addPrimesAndScale (const IndexSet &s1)

  Expand index set by s1, and multiply by Prod_{q in s1}. s1 is disjoint from the current index set, returns log(product).
- void removePrimes (const IndexSet &s1)

  Remove s1 from the index set.
- const FHEcontext & **getContext** () const
- const IndexMap< zz_pX > & **getMap** () const
- const IndexSet & **getIndexSet** () const
- void randomize (const ZZ *seed=NULL)

  Fills each row i with random ints mod pi, uses NTL's PRG.
- void sampleSmall ()

  Coefficients are -1/0/1, Prob[0]=1/2.
- void sampleHWt (long Hwt)

  Coefficients are -1/0/1 with pre-specified number of nonzeros.
- void sampleGaussian (double stdev=0.0)

  Coefficients are Gaussians.
- void toSingleCRT (SingleCRT &scrt, const IndexSet &s) const

  Makes a corresponding SingleCRT object.
- void **toSingleCRT** (SingleCRT &scrt) const
- void **scaleDownToSet** (const IndexSet &s, long ptxtSpace)

Arithmetic operation

Only the "destructive" versions are used, i.e., a += b is implemented but not a + b.

- AltCRT & **Negate** (const AltCRT &other)
- AltCRT & **Negate** ()
- AltCRT & **operator+=** (const AltCRT &other)
- AltCRT & **operator+=** (const ZZX &poly)
- AltCRT & **operator+=** (const ZZ &num)
- AltCRT & **operator+=** (long num)
- AltCRT & **operator-=** (const AltCRT &other)
- AltCRT & **operator-=** (const ZZX &poly)
- AltCRT & **operator-=** (const ZZ &num)
- AltCRT & **operator-=** (long num)
- AltCRT & **operator++** ()
- AltCRT & **operator--** ()
- void **operator++** (int)
- void **operator--** (int)
- AltCRT & **operator*=** (const AltCRT &other)
- AltCRT & **operator*=** (const ZZX &poly)
- AltCRT & **operator*=** (const ZZ &num)
- AltCRT & **operator*=** (long num)
- void **Add** (const AltCRT &other, bool matchIndexSets=true)
- void **Sub** (const AltCRT &other, bool matchIndexSets=true)
- void **Mul** (const AltCRT &other, bool matchIndexSets=true)
- AltCRT & **operator/=** (const ZZ &num)
- AltCRT & **operator/=** (long num)
- void Exp (long k)

  Small-exponent polynomial exponentiation.
- void **automorph** (long k)
- AltCRT & **operator>>=** (long k)

Static Public Member Functions

- static bool setDryRun (bool toWhat=true)

  Used for testing/debugging The dry-run option disables most operations, to save time. This lets us quickly go over the evaluation of a circuit and estimate the resulting noise magnitude, without having to actually compute anything.

Friends

- ostream & **operator**<< (ostream &s, const AltCRT &d)
- istream & **operator**>> (istream &s, AltCRT &d)

5.1.1 Detailed Description

A single-CRT representation of a ring element.

AltCRT offers the same interface as DoubleCRT, but with a different internal representation. That is, polynomials are stored in coefficient representation, modulo each of the small primes in our chain. Currently this class is used only for testing and debugging purposes.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 AltCRT::AltCRT (const ZZX & *poly*, const FHEcontext & *_context*, const IndexSet & *indexSet*)

Parameters

poly	The ring element itself, zero if not specified
_context	The context for this AltCRT object, use "current active context" if not specified
indexSet	Which primes to use for this object, if not specified then use all of them

The documentation for this class was generated from the following files:

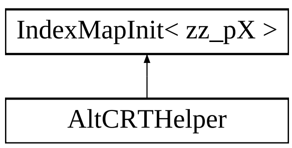
- src/AltCRT.h
- src/AltCRT.cpp

5.2 AltCRTHelper Class Reference

A helper class to enforce consistency within an AltCRT object.

```
#include <AltCRT.h>
```

Inheritance diagram for AltCRTHelper:



Public Member Functions

- **AltCRTHelper** (const FHEcontext &context)
- virtual void init (zz_pX &v)

    the init method ensures that all rows have the same size
- virtual IndexMapInit< zz_pX > ∗ clone () const

    clone allocates a new object and copies the content

5.2.1 Detailed Description

A helper class to enforce consistency within an AltCRT object.

See Section 2.6.2 of the design document (IndexMap)

The documentation for this class was generated from the following file:

- src/AltCRT.h

5.3 Cmod< type > Class Template Reference

template class for both bigint and smallint implementations

`#include <CModulus.h>`

Public Member Functions

- **Cmod** (const Cmod &other)
- **Cmod** (const PAlgebra &zms, const zz &qq, const zz &rt)
- Cmod & **operator=** (const Cmod &other)
- const PAlgebra & **getZMStar** () const
- unsigned **getM** () const
- unsigned **getPhiM** () const
- const zz & **getQ** () const
- const zz & **getRoot** () const
- const zpxModulus & **getPhimX** () const
- zpx & **getScratch** () const
- void restoreModulus () const

    Restore NTL's current modulus.
- void **FFT** (zzv &y, const ZZX &x) const
- void **iFFT** (zpx &x, const zzv &y) const

5.3.1 Detailed Description

template<class type>class Cmod< type >

template class for both bigint and smallint implementations

This is a wrapper around the bluesteinFFT routines, for one modulus q. Two classes are defined here, Cmodulus for a small moduli (long) and CModulus for a large ones (ZZ). These classes are otherwise identical hence they are implemented using a class template.

On initialization, it initizlies NTL's zz_pContext/ZZ_pContext for this q and computes a 2m-th root of unity r mod q and also r^{-1} mod q. Thereafter this class provides FFT and iFFT routines that converts between time & frequency domains. Some tables are computed the first time that each dierctions is called, which are then used in subsequent computations.

The "time domain" polynomials are represented as ZZX, whic are reduced modulo Phi_m(X). The "frequency domain" are jusr vectors of integers (vec_long or vec_ZZ), that store only the evaluation in primitive m-th roots of unity.

The documentation for this class was generated from the following files:

- src/CModulus.h
- src/CModulus.cpp

5.4 CMOD_zz_p Class Reference

typedefs for smallint Cmodulus

```
#include <CModulus.h>
```

Public Types

- typedef long **zz**
- typedef zz_p **zp**
- typedef zz_pX **zpx**
- typedef vec_long **zzv**
- typedef fftRep **fftrep**
- typedef zz_pContext **zpContext**
- typedef zz_pBak **zpBak**
- typedef zz_pXModulus **zpxModulus**

5.4.1 Detailed Description

typedefs for smallint Cmodulus

The documentation for this class was generated from the following file:

- src/CModulus.h

5.5 CMOD_ZZ_p Class Reference

typedefs for bigint CModulus

```
#include <CModulus.h>
```

Public Types

- typedef ZZ **zz**
- typedef ZZ_p **zp**
- typedef ZZ_pX **zpx**
- typedef vec_ZZ **zzv**
- typedef FFTRep **fftrep**
- typedef ZZ_pContext **zpContext**
- typedef ZZ_pBak **zpBak**
- typedef ZZ_pXModulus **zpxModulus**

5.5.1 Detailed Description

typedefs for bigint CModulus

The documentation for this class was generated from the following file:

- src/CModulus.h

5.6 Ctxt Class Reference

A Ctxt object holds a single cipehrtext.

```
#include <Ctxt.h>
```

Public Member Functions

- **Ctxt** (const FHEPubKey &newPubKey, long newPtxtSpace=2)
- Ctxt & **operator=** (const Ctxt &other)
- bool **operator==** (const Ctxt &other) const
- bool **operator!=** (const Ctxt &other) const
- bool **equalsTo** (const Ctxt &other, bool comparePkeys=true) const
- void **clear** ()

Ciphertext arithmetic

- void **negate** ()
- Ctxt & **operator+=** (const Ctxt &other)
- Ctxt & **operator-=** (const Ctxt &other)
- void **addCtxt** (const Ctxt &other, bool negative=false)
- Ctxt & **operator*=** (const Ctxt &other)
- void **automorph** (long k)
- Ctxt & **operator>>=** (long k)
- void smartAutomorph (long k)

  automorphism with re-lienarization
- void **addConstant** (const DoubleCRT &dcrt, double size=0.0)
- void **addConstant** (const ZZX &poly, double size=0.0)
- void **multByConstant** (const DoubleCRT &dcrt, double size=0.0)
- void **multByConstant** (const ZZX &poly, double size=0.0)
- void **multiplyBy** (const Ctxt &other)
- void **multiplyBy2** (const Ctxt &other1, const Ctxt &other2)
- void **square** ()
- void **cube** ()

Ciphertext maintenance

- void **reLinearize** (long keyIdx=0)
- xdouble modSwitchAddedNoiseVar () const

  Estimate the added noise variance.
- void modUpToSet (const IndexSet &s)

  Modulus-switching up (to a larger modulus). Must have primeSet <= s, and s must contain either all the special primes or none of them.
- void modDownToSet (const IndexSet &s)

  Modulus-switching down (to a smaller modulus). mod-switch down to primeSet s, after this call we have prime-Set<=s. s must contain either all special primes or none of them.
- void findBaseSet (IndexSet &s) const

  Fidn the "natural level" of a cipehrtext. Find the highest IndexSet so that mod-switching down to that set results in the dominant noise term being the additive term due to rounding.

Utility methods

- bool inCanonicalForm (long keyID=0) const

  A canonical ciphertext has handles pointing to (1,s)
- bool isCorrect () const

  Would this ciphertext be decrypted without errors?
- const FHEcontext & **getContext** () const
- const FHEPubKey & **getPubKey** () const

- const IndexSet & **getPrimeSet** () const
- const xdouble & **getNoiseVar** () const
- const long **getPtxtSpace** () const
- const long **getKeyID** () const
- const long getLevel () const

  How many primes in the "base-set" for that ciphertext.
- double log_of_ratio () const

  Returns log(noise-variance)/2 - log(q)

Friends

- class **FHEPubKey**
- class **FHESecKey**
- istream & **operator>>** (istream &str, Ctxt &ctxt)
- ostream & **operator<<** (ostream &str, const Ctxt &ctxt)

5.6.1 Detailed Description

A Ctxt object holds a single cipehrtext.

The class Ctxt includes a vector<CtxtPart>: For a Ctxt c, c[i] is the i'th ciphertext part, which can be used also as a DoubleCRT object (since CtxtPart is derived from DoubleCRT). By convention, c[0], the first CtxtPart object in the vector, has skHndl that points to 1 (i.e., it is just added in upon decryption, without being multiplied by anything). We maintain the invariance that all the parts of a ciphertext are defined relative to the same set of primes.

A ciphertext contains also pointers to the general parameters of this FHE instance and the public key, and an estimate of the noise variance. The noise variance is determined by the norm of the canonical embedding of the noise polynomials, namely their evaluations in roots of the ring polynomial (which are the complex primitive roots of unity). We consider each such evaluation point as a random variable, and estimate the variances of these variables. This estimate is heuristic, assuming that various quantities "behave like independent random variables". The variance is added on addition, multiplied on multiplications, remains unchanged for automorphism, and is roughly scaled down by mod-switching with some added factor, and similarly scaled up by key-switching with some added factor. The noiseVar data member of the class keeps the esitmated variance.

The documentation for this class was generated from the following files:

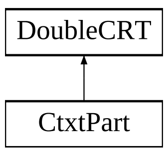
- src/Ctxt.h
- src/Ctxt.cpp

5.7 CtxtPart Class Reference

One entry in a ciphertext vector.

```
#include <Ctxt.h>
```

Inheritance diagram for CtxtPart:



Public Member Functions

- bool **operator==** (const CtxtPart &other) const

- bool **operator!=** (const CtxtPart &other) const
- **CtxtPart** (const FHEcontext &_context)
- **CtxtPart** (const FHEcontext &_context, const IndexSet &s)
- **CtxtPart** (const FHEcontext &_context, const IndexSet &s, const SKHandle &otherHandle)
- **CtxtPart** (const DoubleCRT &other)
- **CtxtPart** (const DoubleCRT &other, const SKHandle &otherHandle)

Public Attributes

- SKHandle skHandle

  The handle is a public data member.

Additional Inherited Members

5.7.1 Detailed Description

One entry in a ciphertext vector.

A cipehrtext part consists of a polynomial (element of the ring R_Q) and a handle to the corresponding secret-key polynomial.

The documentation for this class was generated from the following files:

- src/Ctxt.h
- src/Ctxt.cpp

5.8 Cube Class Reference

Indexing into a hypercube.

5.8.1 Detailed Description

Indexing into a hypercube.

The documentation for this class was generated from the following file:

- src/rotations.cpp

5.9 deep_clone< X > Class Template Reference

Deep copy: initialize with clone.

```
#include <cloned_ptr.h>
```

Static Public Member Functions

- static X ∗ **apply** (const X ∗x)

5.9.1 Detailed Description

template<class X>class deep_clone< X >

Deep copy: initialize with clone.

Template Parameters

X	The class to which this points

The documentation for this class was generated from the following file:

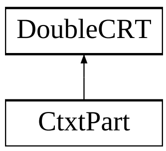
- src/cloned_ptr.h

5.10 DoubleCRT Class Reference

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

```
#include <DoubleCRT.h>
```

Inheritance diagram for DoubleCRT:



Public Member Functions

- DoubleCRT (const ZZX &poly, const FHEcontext &_context, const IndexSet &indexSet)

  Initializing AltCRT from a ZZX polynomial.
- **DoubleCRT** (const ZZX &poly, const FHEcontext &_context)
- DoubleCRT (const ZZX &poly)

  Context is not specified, use the "active context".
- **DoubleCRT** (const FHEcontext &_context)
- DoubleCRT (const FHEcontext &_context, const IndexSet &indexSet)

  Also specify the IndexSet explicitly.
- DoubleCRT & **operator=** (const DoubleCRT &other)
- DoubleCRT & **operator=** (const SingleCRT &other)
- DoubleCRT & **operator=** (const ZZX &poly)
- DoubleCRT & **operator=** (const ZZ &num)
- DoubleCRT & **operator=** (const long num)
- void toPoly (ZZX &p, const IndexSet &s, bool positive=false) const

  Recovering the polynomial in coefficient representation. This yields an integer polynomial with coefficients in [--P/2,P/2], unless the positive flag is set to true, in which case we get coefficients in [0,P-1] (P is the product of all moduli used). Using the optional IndexSet param we compute the polynomial reduced modulo the product of only the ptimes in that set.
- void **toPoly** (ZZX &p, bool positive=false) const
- bool **operator==** (const DoubleCRT &other) const
- bool **operator!=** (const DoubleCRT &other) const
- DoubleCRT & **SetZero** ()
- DoubleCRT & **SetOne** ()
- void breakIntoDigits (vector< DoubleCRT > &dgts, long n) const

Break into n digits,according to the primeSets in context.digits. See Section 3.1.6 of the design document (re-linearization)

- void addPrimes (const IndexSet &s1)

  Expand the index set by s1. It is assumed that s1 is disjoint from the current index set.
- double addPrimesAndScale (const IndexSet &s1)

  Expand index set by s1, and multiply by Prod_{q in s1}. s1 is disjoint from the current index set, returns log(product).
- void removePrimes (const IndexSet &s1)

  Remove s1 from the index set.
- const FHEcontext & **getContext** () const
- const IndexMap< vec_long > & **getMap** () const
- const IndexSet & **getIndexSet** () const
- void randomize (const ZZ ∗seed=NULL)

  Fills each row i with random ints mod pi, uses NTL's PRG.
- void sampleSmall ()

  Coefficients are -1/0/1, Prob[0]=1/2.
- void sampleHWt (long Hwt)

  Coefficients are -1/0/1 with pre-specified number of nonzeros.
- void sampleGaussian (double stdev=0.0)

  Coefficients are Gaussians.
- void toSingleCRT (SingleCRT &scrt, const IndexSet &s) const

  Makes a corresponding SingleCRT object.
- void **toSingleCRT** (SingleCRT &scrt) const
- void **scaleDownToSet** (const IndexSet &s, long ptxtSpace)

Arithmetic operation

Only the "destructive" versions are used, i.e., a += b is implemented but not a + b.

- DoubleCRT & **Negate** (const DoubleCRT &other)
- DoubleCRT & **Negate** ()
- DoubleCRT & **operator+=** (const DoubleCRT &other)
- DoubleCRT & **operator+=** (const ZZX &poly)
- DoubleCRT & **operator+=** (const ZZ &num)
- DoubleCRT & **operator+=** (long num)
- DoubleCRT & **operator-=** (const DoubleCRT &other)
- DoubleCRT & **operator-=** (const ZZX &poly)
- DoubleCRT & **operator-=** (const ZZ &num)
- DoubleCRT & **operator-=** (long num)
- DoubleCRT & **operator++** ()
- DoubleCRT & **operator--** ()
- void **operator++** (int)
- void **operator--** (int)
- DoubleCRT & **operator∗=** (const DoubleCRT &other)
- DoubleCRT & **operator∗=** (const ZZX &poly)
- DoubleCRT & **operator∗=** (const ZZ &num)
- DoubleCRT & **operator∗=** (long num)
- 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)
- DoubleCRT & **operator/=** (long num)
- void Exp (long k)

  Small-exponent polynomial exponentiation.
- void **automorph** (long k)
- DoubleCRT & **operator>>=** (long k)

Static Public Member Functions

- static bool setDryRun (bool toWhat=true)

  Used for testing/debugging The dry-run option disables most operations, to save time. This lets us quickly go over the evaluation of a circuit and estimate the resulting noise magnitude, without having to actually compute anything.

Friends

- ostream & **operator**<< (ostream &s, const DoubleCRT &d)
- istream & **operator**>> (istream &s, DoubleCRT &d)

5.10.1 Detailed Description

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

Double-CRT form is a matrix of L rows and phi(m) columns. The i'th row contains the FFT of the element wrt the ith prime, i.e. the evaluations of the polynomial at the primitive mth roots of unity mod the ith prime. The polynomial thus represented is defined modulo the product of all the primes in use.

The list of primes is defined by the data member indexMap. indexMap.getIndexSet() defines the set of indices of primes associated with this DoubleCRT object: they index the primes stored in the associated FHEContext.

Arithmetic operations are computed modulo the product of the primes in use and also modulo Phi_m(X). Arithmetic operations can only be applied to DoubleCRT objects relative to the same context, trying to add/multiply objects that have different FHEContext objects will raise an error.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 DoubleCRT::DoubleCRT (const ZZX & *poly*, const FHEcontext & *_context*, const IndexSet & *indexSet*)

Initializing AltCRT from a ZZX polynomial.

Parameters

poly	The ring element itself, zero if not specified
_context	The context for this AltCRT object, use "current active context" if not specified
indexSet	Which primes to use for this object, if not specified then use all of them

The documentation for this class was generated from the following files:

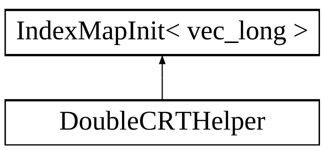
- src/DoubleCRT.h
- src/DoubleCRT.cpp

5.11 DoubleCRTHelper Class Reference

A helper class to enforce consistency within an DoubleCRTHelper object.

```
#include <DoubleCRT.h>
```

Inheritance diagram for DoubleCRTHelper:



Public Member Functions

- **DoubleCRTHelper** (const FHEcontext &context)
- virtual void init (vec_long &v)

  the init method ensures that all rows have the same size
- virtual IndexMapInit< vec_long > ∗ clone () const

  clone allocates a new object and copies the content

5.11.1 Detailed Description

A helper class to enforce consistency within an DoubleCRTHelper object.

See Section 2.6.2 of the design document (IndexMap)

The documentation for this class was generated from the following file:

- src/DoubleCRT.h

5.12 EncryptedArray Class Reference

A simple wrapper for a smart pointer to an EncryptedArrayBase. This is the interface that higher-level code should use.

```
#include <EncryptedArray.h>
```

Public Member Functions

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

  constructor: G defaults to the monomial X
- template<class type >
  const EncryptedArrayDerived
  < type > & getDerived (type) const

  downcast operator example: const EncryptedArrayDerived<PA_GF2>& rep = ea.getDerived(PA_GF2());

Direct access to EncryptedArrayBase methods

- const FHEcontext & **getContext** () const
- const long **getDegree** () const
- void **rotate** (Ctxt &ctxt, long k) const
- void **shift** (Ctxt &ctxt, long k) const
- void **rotate1D** (Ctxt &ctxt, long i, long k, bool dc=false) const
- void **shift1D** (Ctxt &ctxt, long i, long k) const
- 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
- void **encodeUnitSelector** (ZZX &ptxt, long i) const
- 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
- 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
- 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
- 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
- void **buildLinPolyCoeffs** (vector< ZZX > &C, const vector< ZZX > &L) const
- long **size** () const
- long **dimension** () const
- long **sizeOfDimension** (long i) const
- long **nativeDimension** (long i) const
- long **coordinate** (long i, long k) const

5.12.1 Detailed Description

A simple wrapper for a smart pointer to an EncryptedArrayBase. This is the interface that higher-level code should use.

The documentation for this class was generated from the following file:

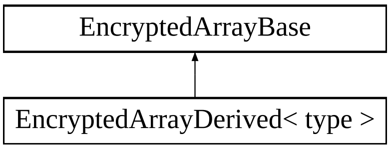
- src/EncryptedArray.h

5.13 EncryptedArrayBase Class Reference

virtual class for data-movement operations on arrays of slots

```
#include <EncryptedArray.h>
```

Inheritance diagram for EncryptedArrayBase:



Public Member Functions

- virtual EncryptedArrayBase ∗ **clone** () const =0
- virtual const FHEcontext & **getContext** () const =0
- virtual const long **getDegree** () const =0
- virtual void rotate (Ctxt &ctxt, long k) const =0

  Rotation/shift as a linear array.
- virtual void shift (Ctxt &ctxt, long k) const =0

  Non-cyclic shift with zero fill.
- virtual void rotate1D (Ctxt &ctxt, long i, long k, bool dc=false) const =0

  rotate k positions along the i'th dimension
- virtual void shift1D (Ctxt &ctxt, long i, long k) const =0

  Shift k positions along the i'th dimension with zero fill.
- virtual void buildLinPolyCoeffs (vector< ZZX > &C, const vector< ZZX > &L) const =0

  Linearized polynomials. L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..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,.
- long size () const

  Total size (# of slots) of hypercube.
- long dimension () const

  Number of dimensions of hypercube.
- long sizeOfDimension (long i) const

  Size of given dimension.

- long nativeDimension (long i) const

  Is rotations in given dimension a "native" operation?
- long coordinate (long i, long k) const

  returns coordinate of index k along the i'th dimension

Encoding/decoding methods

- virtual void **encode** (ZZX &ptxt, const vector< long > &array) const =0
- virtual void **encode** (ZZX &ptxt, const vector< ZZX > &array) const =0
- virtual void **encode** (ZZX &ptxt, const PlaintextArray &array) const =0
- virtual void **decode** (vector< long > &array, const ZZX &ptxt) const =0
- virtual void **decode** (vector< ZZX > &array, const ZZX &ptxt) const =0
- virtual void **decode** (PlaintextArray &array, const ZZX &ptxt) const =0
- virtual void encodeUnitSelector (ZZX &ptxt, long i) const =0

  Encodes a vector with 1 at position i and 0 everywhere else.

Encoding+encryption/decryption+decoding

- virtual void **encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< long > &ptxt) const =0
- virtual void **encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< ZZX > &ptxt) const =0
- virtual void **encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const PlaintextArray &ptxt) const =0
- virtual void **decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< long > &ptxt) const =0
- virtual void **decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< ZZX > &ptxt) const =0
- virtual void **decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, PlaintextArray &ptxt) const =0
- virtual void select (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< long > &selector) const =0

  *MUX: ctxt1 = ctxt1*selector + ctxt2*(1-selector)*
- virtual void **select** (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< ZZX > &selector) const =0
- virtual void **select** (Ctxt &ctxt1, const Ctxt &ctxt2, const PlaintextArray &selector) const =0

5.13.1 Detailed Description

virtual class for data-movement operations on arrays of slots

An object ea of type EncryptedArray stores information about an FHEcontext context, and a monic polynomial G. If context defines parameters m, p, and r, then ea is a helper abject that supports encoding/decoding and encryption/decryption of vectors of plaintext slots over the ring (Z/(p^r)[X])/(G).

The polynomial G should be irreducble over Z/(p^r) (this is not checked). The degree of G should divide the multiplicative order of p modulo m (this is checked). Currently, the following restriction is imposed:

either r == 1 or deg(G) == 1 or G == factors[0].

ea stores objects in the polynomial the polynomial ring Z/(p^r)[X].

Just as for the class PAlegebraMod, if p == 2 and r == 1, then these polynomials are represented as GF2X's, and otherwise as zz_pX's. Thus, the types of these objects are not determined until run time. As such, we need to use a class heirarchy, which mirrors that of PAlgebraMod, as follows.

EncryptedArrayBase is a virtual class

EncryptedArrayDerived<type> is a derived template class, where type is either PA_GF2 or PA_zz_p.

The class EncryptedArray is a simple wrapper around a smart pointer to an EncryptedArrayBase object: copying an EncryptedArray object results is a "deep copy" of the underlying object of the derived class.

5.13.2 Member Function Documentation

5.13.2.1 virtual void EncryptedArrayBase::buildLinPolyCoeffs (vector< ZZX > & *C*, const vector< ZZX > & *L*) const [pure virtual]

Linearized polynomials. L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..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) mod G) = sum_{i=0}^{d-1} (C[j] mod G) * (h(X^{p^j}) mod G).

Implemented in EncryptedArrayDerived< type >.

5.13.2.2 virtual void EncryptedArrayBase::rotate1D (Ctxt & *ctxt*, long *i*, long *k*, bool *dc* = `false`) const [pure virtual]

rotate k positions along the i'th dimension

Parameters

dc	means "don't care", which means that the caller guarantees that only zero elements rotate off the end – this allows for some optimizations that would not otherwise be possible

Implemented in EncryptedArrayDerived< type >.

The documentation for this class was generated from the following file:

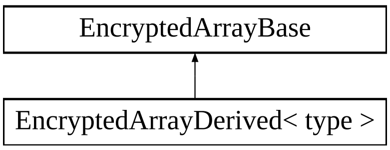
- src/EncryptedArray.h

5.14 EncryptedArrayDerived< type > Class Template Reference

Derived concrete implementation of EncryptedArrayBase.

```
#include <EncryptedArray.h>
```

Inheritance diagram for EncryptedArrayDerived< type >:



Public Member Functions

- **EncryptedArrayDerived** (const FHEcontext &_context, const RX &_G=RX(1, 1))
- **EncryptedArrayDerived** (const EncryptedArrayDerived &other)
- EncryptedArrayDerived & **operator=** (const EncryptedArrayDerived &other)
- virtual EncryptedArrayBase * **clone** () const
- const RX & **getG** () const
- virtual const FHEcontext & **getContext** () const
- virtual const long **getDegree** () const
- virtual void rotate (Ctxt &ctxt, long k) const

  Rotation/shift as a linear array.
- virtual void shift (Ctxt &ctxt, long k) const

  Non-cyclic shift with zero fill.
- virtual void rotate1D (Ctxt &ctxt, long i, long k, bool dc=false) const

  rotate k positions along the i'th dimension
- virtual void shift1D (Ctxt &ctxt, long i, long k) const

  Shift k positions along the i'th dimension with zero fill.
- virtual void **encode** (ZZX &ptxt, const vector< long > &array) const
- virtual void **encode** (ZZX &ptxt, const vector< ZZX > &array) const
- virtual void **encode** (ZZX &ptxt, const PlaintextArray &array) const

- virtual void encodeUnitSelector (ZZX &ptxt, long i) const

  Encodes a vector with 1 at position i and 0 everywhere else.
- virtual void **decode** (vector< long > &array, const ZZX &ptxt) const
- virtual void **decode** (vector< ZZX > &array, const ZZX &ptxt) const
- virtual void **decode** (PlaintextArray &array, const ZZX &ptxt) const
- virtual void **encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< long > &ptxt) const
- virtual void **encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< ZZX > &ptxt) const
- virtual void **encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const PlaintextArray &ptxt) const
- virtual void **decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< long > &ptxt) const
- virtual void **decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< ZZX > &ptxt) const
- virtual void **decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, PlaintextArray &ptxt) const
- virtual void select (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< long > &selector) const

  MUX: ctxt1 = ctxt1∗selector + ctxt2∗(1-selector)
- virtual void **select** (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< ZZX > &selector) const
- virtual void **select** (Ctxt &ctxt1, const Ctxt &ctxt2, const PlaintextArray &selector) const
- void **encode** (ZZX &ptxt, const vector< RX > &array) const
- void **decode** (vector< RX > &array, const ZZX &ptxt) const
- void **encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< RX > &ptxt) const
- void **decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< RX > &ptxt) const
- void buildLinPolyCoeffs (vector< ZZX > &C, const vector< ZZX > &L) const

  Linearized polynomials. L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..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,.

5.14.1 Detailed Description

template<class type>class EncryptedArrayDerived< type >

Derived concrete implementation of EncryptedArrayBase.

5.14.2 Member Function Documentation

5.14.2.1 template<class type > void EncryptedArrayDerived< type >::buildLinPolyCoeffs (vector< ZZX > & *C*, const vector< ZZX > & *L*) const [virtual]

Linearized polynomials. L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..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) mod G) = sum_{i=0}^{d-1} (C[j] mod G) ∗ (h(X^{p^j}) mod G).

Implements EncryptedArrayBase.

5.14.2.2 template<class type > void EncryptedArrayDerived< type >::rotate1D (Ctxt & *ctxt*, long *i*, long *k*, bool *dc* = false) const [virtual]

rotate k positions along the i'th dimension

Parameters

dc	means "don't care", which means that the caller guarantees that only zero elements rotate off the end – this allows for some optimizations that would not otherwise be possible

Implements EncryptedArrayBase.

The documentation for this class was generated from the following files:

- src/EncryptedArray.h
- src/EncryptedArray.cpp

5.15 FHEcontext Class Reference

Maintaining the parameters.

`#include <FHEContext.h>`

Public Member Functions

- **FHEcontext** (unsigned m, unsigned p, unsigned r)
- bool **operator==** (const FHEcontext &other) const
- bool **operator!=** (const FHEcontext &other) const
- long ithPrime (unsigned i) const

  The ith small prime in the modulus chain.
- const Cmodulus & ithModulus (unsigned i) const

  Cmodulus object corresponding to ith small prime in the chain.
- long numPrimes () const

  Total number of small prime in the chain.
- bool isZeroDivisor (const ZZ &num) const

  Is num divisible by any of the primes in the chain?
- bool inChain (long p) const

  Is p already in the chain?
- double logOfPrime (unsigned i) const

  Returns the natural logarithm of the ith prime.
- double logOfProduct (const IndexSet &s) const

  Returns the natural logarithm of productOfPrimes(s)
- void AddPrime (long p, bool special)

  Add p to the chain, if it's not already there.

- void productOfPrimes (ZZ &p, const IndexSet &s) const

  The product of all the primes in the given set.
- ZZ **productOfPrimes** (const IndexSet &s) const

Public Attributes

- PAlgebra zMStar

  The structure of Zm∗.
- PAlgebraMod alMod

  The structure of Z[X]/(Phi_m(X),2)
- xdouble stdev

  sqrt(variance) of the LWE error (default=3.2)
- IndexSet ctxtPrimes

  The "ciphertext primes", used for fresh ciphertexts.
- IndexSet specialPrimes

  All the other primes in the chain.
- vector< IndexSet > digits

  The set of primes for the digits.

Friends

I/O routines

To write out all the data associated with a context, do the following:

```
writeContextBase(str, context);
str << context;
```

The first function call writes out just [m p r], which is the data needed to invoke the context constructor.

The second call writes out all other information, including the stdev field, the prime sequence (including which primes are "special"), and the digits info.

To read in all the data associated with a context, do the following:

```
unsigned m, p, r;
readContextBase(str, m, p, r);

FHEcontext context(m, p, r);

str >> context;
```

The call to readContextBase just reads the values m, p, r. Then, after constructing the context, the >> operator reads in and attaches all other information.

- void writeContextBase (ostream &str, const FHEcontext &context)
  write [m p r] data
- ostream & operator<< (ostream &str, const FHEcontext &context)
  Write all other data.
- void readContextBase (istream &str, unsigned &m, unsigned &p, unsigned &r)
  read [m p r] data, needed to construct context
- istream & operator>> (istream &str, FHEcontext &context)
  read all other data associated with context

5.15.1 Detailed Description

Maintaining the parameters.

5.15.2 Member Data Documentation

5.15.2.1 IndexSet FHEcontext::ctxtPrimes

The "ciphertext primes", used for fresh ciphertexts.

The public encryption key and "fresh" ciphertexts are encrypted relative to only a subset of the primes, to allow for mod-UP during key-switching. See section 3.1.6 in the design document (key-switching). In ctxtPrimes we keep the indexes of this subset. Namely, for a ciphertext part p in a fresh ciphertext we have p.getMap().getIndexSet()==ctxtPrimes.

5.15.2.2 vector<IndexSet> FHEcontext::digits

The set of primes for the digits.

The different columns in any key-switching matrix contain encryptions of multiplies of the secret key, sk, B1∗sk, B2∗B1∗sk, B3∗B2∗B1∗sk,... with each Bi a product of a few "non-special" primes in the chain. The digits data member indicate which primes correspond to each of the Bi's. These are all IndexSet objects, whose union is the subset ctxtPrimes.

The number of Bi's is one less than the number of columns in the key switching matrices (since the 1st column encrypts sk, without any Bi's), but we keep in the digits vector also an entry for the primes that do not participate in any Bi (so digits.size() is the same as the number of columns in the key switching matrices). See section 3.1.6 in the design document (key-switching).

5.15.2.3 IndexSet FHEcontext::specialPrimes

All the other primes in the chain.

For convenience, we also keep in specialPrimes the complemeting subset, i.e., specialPrimes = [0,numPrimes()-1] setminus ctxtPrimes.

The documentation for this class was generated from the following files:

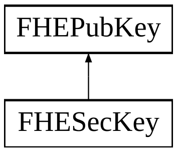
- src/FHEContext.h
- src/FHEContext.cpp

5.16 FHEPubKey Class Reference

The public key.

```
#include <FHE.h>
```

Inheritance diagram for FHEPubKey:



Public Member Functions

- **FHEPubKey** (const FHEcontext &_context)
- void **clear** ()
- bool **operator==** (const FHEPubKey &other) const
- bool **operator!=** (const FHEPubKey &other) const
- const FHEcontext & **getContext** () const
- long getSKeyWeight (long keyID=0) const

  The Hamming weight of the secret key.
- bool isReachable (long k, long keyID=0) const

  Is it possible to re-linearize the automorphism X -> X^k See Section 3.2.2 in the design document (KeySwitchMap)
- void setKeySwitchMap (long keyId=0)

  Compute the reachability graph of key-switching matrices See Section 3.2.2 in the design document (KeySwitchMap)
- long Encrypt (Ctxt &ciphertxt, const ZZX &plaintxt, long ptxtSpace=0) const

  Result returned in the ciphertext argument, The resurn value is the plaintext-space for that ciphertext.

Find key-switching matrices

- const KeySwitch & getKeySWmatrix (const SKHandle &from, long toID=0) const

  Find a key-switching matrix by its indexes. If no such matrix exists it returns a dummy matrix with toKeyID==-1.
- const KeySwitch & **getKeySWmatrix** (long fromSPower, long fromXPower, long fromID=0, long toID=0) const
- bool **haveKeySWmatrix** (const SKHandle &from, long toID=0) const
- bool **haveKeySWmatrix** (long fromSPower, long fromXPower, long fromID=0, long toID=0) const
- const KeySwitch & getAnyKeySWmatrix (const SKHandle &from) const

  Is there a matrix from this key to any base key?
- bool **haveAnyKeySWmatrix** (const SKHandle &from) const
- const KeySwitch & getNextKSWmatrix (long fromXPower, long fromID=0) const

  Get the next matrix to use for multi-hop automorphism See Section 3.2.2 in the design document.

Friends

- class **FHESecKey**
- ostream & **operator**<< (ostream &str, const FHEPubKey &pk)
- istream & **operator**>> (istream &str, FHEPubKey &pk)

5.16.1 Detailed Description

The public key.

The documentation for this class was generated from the following files:

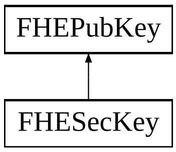
- src/FHE.h
- src/FHE.cpp

5.17 FHESecKey Class Reference

The secret key.

```
#include <FHE.h>
```

Inheritance diagram for FHESecKey:



Public Member Functions

- **FHESecKey** (const FHEcontext &_context)
- bool **operator==** (const FHESecKey &other) const
- bool **operator!=** (const FHESecKey &other) const
- void **clear** ()
- long ImportSecKey (const DoubleCRT &sKey, long hwt, long ptxtSpace=0)
- long GenSecKey (long hwt, long ptxtSpace=0)
- void GenKeySWmatrix (long fromSPower, long fromXPower, long fromKeyIdx=0, long toKeyIdx=0, long ptxt-Space=0)
- void **Decrypt** (ZZX &plaintxt, const Ctxt &ciphertxt) const
- void Decrypt (ZZX &plaintxt, const Ctxt &ciphertxt, ZZX &f) const

  Debugging version, returns in f the polynomial before reduction modulo the ptxtSpace.
- long Encrypt (Ctxt &ctxt, const ZZX &ptxt, long ptxtSpace=0, long skIdx=0) const

  Symmetric encryption using the secret key.

Public Attributes

- vector< DoubleCRT > **sKeys**

Friends

- ostream & **operator**<< (ostream &str, const FHESecKey &sk)
- istream & **operator**>> (istream &str, FHESecKey &sk)

5.17.1 Detailed Description

The secret key.

5.17.2 Member Function Documentation

5.17.2.1 void FHESecKey::GenKeySWmatrix (long *fromSPower*, long *fromXPower*, long *fromKeyIdx* = 0, long *toKeyIdx* = 0, long *ptxtSpace* = 0)

Generate a key-switching matrix and store it in the public key. The i'th column of the matrix encrypts fromKey∗B1∗-B2∗...∗B{i-1}∗Q under toKey, relative to the largest modulus (i.e., all primes) and plaintext space p. Q is the product of special primes, and the Bi's are the products of primes in the i'th digit. The plaintext space defaults to 2^r, as defined by context.mod2r.

5.17.2.2 long FHESecKey::GenSecKey (long *hwt*, long *ptxtSpace* = 0) [inline]

Key generation: This procedure generates a single secret key, pushes it onto the sKeys list using ImportSecKey from above.

5.17.2.3 long FHESecKey::ImportSecKey (const DoubleCRT & *sKey*, long *hwt*, long *ptxtSpace* = 0)

We allow the calling application to choose a secret-key polynomial by itself, then insert it into the FHESecKey object, getting the index of that secret key in the sKeys list. If this is the first secret-key for this object then the procedure below also generate a corresponding public encryption key. It is assumed that the context already contains all parameters.

The documentation for this class was generated from the following files:

- src/FHE.h
- src/FHE.cpp

5.18 FHEtimer Class Reference

A simple class to toggle timing information on and off.

Public Attributes

- bool **isOn**
- clock_t **counter**
- long **numCalls**

5.18.1 Detailed Description

A simple class to toggle timing information on and off.

The documentation for this class was generated from the following file:

- src/timing.cpp

5.19 IndexMap< T > Class Template Reference

IndexMap<T> implements a generic map indexed by a dynamic index set.

`#include <IndexMap.h>`

Public Member Functions

- IndexMap ()

  The empty map.
- IndexMap (IndexMapInit< T > *_init)

  A map with an initialization object. This associates a method for initializing new elements in the map. 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 _init->init(t) is called (t is passed by reference). To use this feature, you need to derive a subclass of IndexMapInit<T> that defines the init function. This "helper object" should be created using operator new, and the pointer is "exclusively owned" by the map object.
- const IndexSet & getIndexSet () const

  Get the underlying index set.
- T & operator[] (long j)

  Access functions: will raise an error if j does not belong to the current index set.
- const T & **operator[]** (long j) const
- void insert (long j)

  Insert indexes to the IndexSet. Insertion will cause new T objects to be created, using the default constructor, and possibly initilized via the IndexMapInit<T> pointer.
- void **insert** (const IndexSet &s)
- void remove (long j)

  Delete indexes from IndexSet, may cause objects to be destroyed.
- void **remove** (const IndexSet &s)
- void **clear** ()

5.19.1 Detailed Description

template<class T>class IndexMap< T >

IndexMap<T> implements a generic map indexed by a dynamic index set.

Additionally, it allows new elements of the map to be initialized in a flexible manner.

The documentation for this class was generated from the following file:

- src/IndexMap.h

5.20 IndexMapInit< T > Class Template Reference

Initializing elements in an IndexMap.

`#include <IndexMap.h>`

Public Member Functions

- virtual void init (T &)=0

  Initialization function, override with initialization code.
- virtual IndexMapInit< T > * clone () const =0

  Cloning a pointer, override with code to create a fresh copy.

5.20.1 Detailed Description

template<class T>class IndexMapInit< T >

Initializing elements in an IndexMap.

The documentation for this class was generated from the following file:

- src/IndexMap.h

5.21 IndexSet Class Reference

A dynamic set of non-negative integers.

`#include <IndexSet.h>`

Public Member Functions

- **IndexSet** (long low, long high)
- **IndexSet** (long j)
- long first () const

  Returns the first element, 0 if the set is empty.
- long last () const

  Returns the last element, -1 if the set is empty.
- long next (long j) const

  Returns the next element after j, if any; otherwise j+1.
- long **prev** (long j) const
- long card () const

  The cardinality of the set.
- bool contains (long j) const

  Returns true iff the set contains j.
- bool contains (const IndexSet &s) const

  Returns true iff the set contains s.
- bool disjointFrom (const IndexSet &s) const

  Returns true iff the set is disjoint from s.
- bool **operator==** (const IndexSet &s) const
- bool **operator!=** (const IndexSet &s) const
- void clear ()

  Set to the empty set.
- void insert (long j)

  Add j to the set.
- void remove (long j)

  Remove j from the set.
- void insert (const IndexSet &s)

  Add s to the set (union)
- void remove (const IndexSet &s)

  Remove s from the set (set minus)
- void retain (const IndexSet &s)

  Retains only those elements that are also in s (intersection)

Static Public Member Functions

- static const IndexSet & emptySet ()

  Read-only access to an empty set.

5.21.1 Detailed Description

A dynamic set of non-negative integers.

You can iterate through a set as follows:

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

The documentation for this class was generated from the following files:

- src/IndexSet.h
- src/IndexSet.cpp

5.22 KeySwitch Class Reference

Key-switching matrices.

`#include <FHE.h>`

Public Member Functions

- **KeySwitch** (long sPow=0, long xPow=0, long fromID=0, long toID=0, long p=0)
- **KeySwitch** (const SKHandle &_fromKey, long fromID=0, long toID=0, long p=0)
- bool **operator==** (const KeySwitch &other) const
- bool **operator!=** (const KeySwitch &other) const
- unsigned **NumCols** () const
- void verify (FHESecKey &sk)

  A debugging method.
- void readMatrix (istream &str, const FHEcontext &context)

  Read a key-switching matrix from input.

Static Public Member Functions

- static const KeySwitch & dummy ()

  returns a dummy static matrix with toKeyId == -1

Public Attributes

- SKHandle **fromKey**
- long **toKeyID**
- long **ptxtSpace**
- vector< DoubleCRT > **b**
- ZZ **prgSeed**

5.22.1 Detailed Description

Key-switching matrices.

There are basically two approaches for how to do key-switching: either decompose the mod-q ciphertext into bits (or digits) to make it low-norm, or perform the key-switching operation mod Q>>q. The tradeoff is that when decomposing the (coefficients of the) ciphertext into t digits, we need to increase the size of the key-switching matrix by a factor of t (and the running time similarly grows). On the other hand if we do not decompose at all then we need to work modulo Q>q^2, which means that the bitsize of our largest modulus q0 more than doubles (and hence also the parameter m more than doubles). In general if we decompose into digits of size B then we need to work with Q>q*B.)

The part of the spectrum where we expect to find the sweet spot is when we decompose the ciphertext into digits of size B=q0^{1/t} for some small constant t (maybe t=2,3 or so). This means that our largest modulus has to be Q>q0^{1+1/t}, which increases also the parameter m by a factor (1+1/t). It also means that for key-switching in the top levels we would break the ciphertext to t digits, hence the key-switching matrix will have t columns.

A key-switch matrix W[s'->s] converts a ciphertext-part with respect to secret-key polynomial s' into a canonical cipehrtext (i.e. a two-part ciphertext with respect to (1,s)). The matrix W is a 2-by-t matrix of DoubleCRT objects. The bottom row are just (psudo)random elements. Then for column i, if the bottom element is ai then the top element is set as bi = P*Bi*s' + p*ei - s ai mod P*q0, where p is the plaintext space (i.e. 2 or 2^r) and Bi is the product of the digits-sizes corresponding to columns 0...i-1. (For example if we have digit sizes 3,5,7 then B0=1, B1=3, B2=15 and B3=105.) Also, q0 is the product of all the "ciphertext primes" and P is roughly the product of all the special primes. (Actually, if Q is the product of all the special primes then P=Q*(Q^{-1} mod p).)

In this implementation we save some space, by keeping only a PRG seed for generating the pseudo-random elements, rather than the elements themselves.

To convert a cipehrtext part R, we break R into digits R= Bi Ri, then set (q0,q1)^T = Ri * column-i. Note that we have (1,s) (q0,q1) = Ri*(s ai-s ai+p*ei+P*Bi*s') = P * Bi*Ri s' + p Ri ei = P * R s' + p*a-small-element (mod P*q0) where the last element is small since the ei's are small and |Ri|<B. Note that if the ciphertext is encrypted relative to plaintext space p' and then key-switched with matrices W relative to plaintext space p, then we get a mew ciphertxt with noise p'*small+p*small, so it is valid relative to plaintext space GCD(p',p).

The matrix W is defined modulo Q>t*B*sigma*q0 (with sigma a bound on the size of the ei's), and Q is the product of all the small primes in our moduli chain. However, if p is much smaller than B then is is enough to use W mod Qi with Qi a smaller modulus, Q>p*sigma*q0. Also note that if p<Br then we will be using only first r columns of the matrix W.

The documentation for this class was generated from the following files:

- src/FHE.h
- src/FHE.cpp

5.23 MappingData< type > Class Template Reference

Auxilliary structure to support encoding/decoding slots.

```
#include <PAlgebra.h>
```

Public Member Functions

- const RX & **getG** () const
- long **getDegG** () const

Friends

- class **PAlgebraModDerived**< **type** >

5.23.1 Detailed Description

template<class type>class MappingData< type >

Auxilliary structure to support encoding/decoding slots.

The documentation for this class was generated from the following file:

- src/PAlgebra.h

5.24 PAlgebra Class Reference

The structure of (Z/mZ)∗ /(p)

```
#include <PAlgebra.h>
```

Public Member Functions

- **PAlgebra** (unsigned mm, unsigned pp=2)
- bool **operator==** (const PAlgebra &other) const
- bool **operator!=** (const PAlgebra &other) const
- void printout () const

  Prints the structure in a readable form.
- unsigned getM () const

  Returns m.
- unsigned getP () const

  Returns p.
- unsigned getPhiM () const

  Returns phi(m)
- unsigned getOrdP () const

  The order of p in (Z/mZ)^∗.
- unsigned getNSlots () const

  The number of plaintext slots = phi(m) = ord(p)
- const ZZX & getPhimX () const

  The cyclotomix polynomial Phi_m(X)
- unsigned numOfGens () const

  The number of generators in (Z/mZ)^∗ /(p)
- unsigned ZmStarGen (unsigned i) const

  the i'th generator in (Z/mZ)^∗ /(p) (if any)
- unsigned OrderOf (unsigned i) const

  The order of i'th generator (if any)
- bool SameOrd (unsigned i) const

  Is ord(i'th generator) the same as its order in (Z/mZ)^∗?

Translation between index, represnetatives, and exponents

- unsigned ith_rep (unsigned i) const

  Returns the i'th element in T.
- int indexOfRep (unsigned t) const

  Returns the index of t in T.
- bool isRep (unsigned t) const

  Is t in T?
- int indexInZmstar (unsigned t) const

Returns the index of t in (Z/mZ)∗.
- bool inZmStar (unsigned t) const

  Is t in [0,m-1] with (t,m)=1?
- long coordinate (long i, long k) const

  Returns ith coordinate of index k along the i'th dimension. See Section 2.4 in the design document.
- unsigned exponentiate (const vector< unsigned > &exps, bool onlySameOrd=false) const

  Returns prod_i gi^{exps[i]} mod m. If onlySameOrd=true, use only generators that have the same order as in (Z/mZ)^∗.
- const int ∗ dLog (unsigned t) const

  Inverse of exponentiate.
- unsigned qGrpOrd (bool onlySameOrd=false) const
- bool nextExpVector (vector< unsigned > &exps) const

5.24.1 Detailed Description

The structure of (Z/mZ)∗ /(p)

A PAlgebra object is determined by an integer m and a prime p, where p does not divide m. It holds information descrtibing the structure of (Z/mZ)^∗, which is isomorphic to the Galois group over A = Z[X]/Phi_m(X)).

We represent (Z/mZ)^∗ as (Z/mZ)^∗ = (p) x (g1,g2,...) x (h1,h2,...) where the group generated by g1,g2,... consists of the elements that have the same order in (Z/mZ)^∗ as in (Z/mZ)^∗ /(p,g_1,...,g_{i-1}), and h1,h2,... generate the remaining quotient group (Z/mZ)^∗ /(p,g1,g2,...).

We let T subset (Z/mZ)^∗ be a set of representatives for the quotient group (Z/mZ)^∗ /(p), defined as T={ prod_i gi^{ei} ∗ prod_j hj^{ej} } where the ei's range over 0,1,...,ord(gi)-1 and the ej's range over 0,1,...ord(hj)-1 (these last orders are in (Z/mZ)^∗ /(p,g1,g2,...)).

Phi_m(X) is factored as Phi_m(X)= prod_{t in T} F_t(X) mod p, where the F_t's are irreducible modulo p. An arbitrary factor is chosen as F_1, then for each t in T we associate with the index t the factor F_t(X) = GCD(F_1(X^t), Phi_-m(X)).

Note that fixing a representation of the field R=(Z/pZ)[X]/F_1(X) and letting z be a root of F_1 in R (which is a primitive m-th root of unity in R), we get that F_t is the minimal polynomial of z^{1/t}.

5.24.2 Member Function Documentation

5.24.2.1 bool PAlgebra::nextExpVector (vector< unsigned > & *exps*) const

exps is an array of exponents (the dLog of some t in T), this function incerement exps lexicographic order, reutrn false if it cannot be incremented (because it is at its maximum value)

5.24.2.2 unsigned PAlgebra::qGrpOrd (bool *onlySameOrd* = `false`) const `[inline]`

The order of the quoteint group (Z/mZ)^∗ /(p) (if flag=false), or the subgroup of elements with the same order as in (Z/mZ)^∗ (if flag=true)

The documentation for this class was generated from the following files:

- src/PAlgebra.h
- src/PAlgebra.cpp

5.25 PAlgebraMod Class Reference

The structure of Z[X]/(Phi_m(X), p)

`#include <PAlgebra.h>`

Public Member Functions

- **PAlgebraMod** (const PAlgebra &zMStar, long r)
- template<class type >
  const PAlgebraModDerived< type > & getDerived (type) const
- bool **operator==** (const PAlgebraMod &other) const
- bool **operator!=** (const PAlgebraMod &other) const
- PA_tag getTag () const

  Returns the type tag: PA_GF2_tag or PA_zz_p_tag.
- const PAlgebra & getZMStar () const

  Returns reference to underlying PAlgebra object.
- const vector< ZZX > & getFactorsOverZZ () const

  Returns reference to the factorization of Phi_m(X) mod p^r, but as ZZX's.
- long getR () const

  The value r.
- long getPPowR () const

  The value p^r.
- void restoreContext () const

  Restores the NTL context for p^r.
- void genMaskTable () const

  Generates the "mask table" that is used to support rotations.

5.25.1 Detailed Description

The structure of Z[X]/(Phi_m(X), p)

An object of type PAlgebraMod stores information about a PAlgebra object zMStar, and an integer r. It also provides support for encoding and decoding plaintext slots.

the PAlgebra object zMStar defines (Z/mZ)^∗ /(0), and the PAlgebraMod object stores various tables related to the polynomial ring Z/(p^r)[X]. To do this most efficiently, if p == 2 and r == 1, then these polynomials are represented as GF2X's, and otherwise as zz_pX's. Thus, the types of these objects are not determined until run time. As such, we need to use a class heirarchy, as follows.

- PAlgebraModBase is a virtual class

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

- The class PAlgebraMod is a simple wrapper around a smart pointer to a PAlgebraModBase object: copying a PAlgebra object results is a "deep copy" of the underlying object of the derived class. It provides dDirect access to the virtual methods of PAlgebraModBase, along with a "downcast" operator to get a reference to the object as a derived type, and also == and != operators.

5.25.2 Member Function Documentation

5.25.2.1 void PAlgebraMod::genMaskTable ( ) const [inline]

Generates the "mask table" that is used to support rotations.

maskTable[i][j] is a polynomial representation of a mask that is 1 in all slots whose i'th coordinate is at least j, and 0 elsewhere. We have:

```
maskTable.size() == zMStar.numOfGens()     // # of generators
for i = 0..maskTable.size()-1:
  maskTable[i].size() == zMStar.OrderOf(i) // order of generator i
```

5.25.2.2 template<class type > const PAlgebraModDerived<type>& PAlgebraMod::getDerived (type) const [inline]

Downcast operator example: const PAlgebraModDerived<PA_GF2>& rep = alMod.getDerived(PA_GF2());

The documentation for this class was generated from the following file:

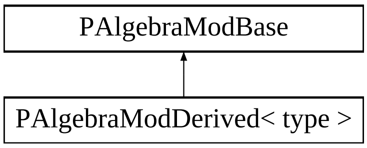
- src/PAlgebra.h

5.26 PAlgebraModBase Class Reference

Virtual base class for PAlgebraMod.

`#include <PAlgebra.h>`

Inheritance diagram for PAlgebraModBase:



Public Member Functions

- virtual PAlgebraModBase * **clone** () const =0
- virtual PA_tag getTag () const =0

  Returns the type tag: PA_GF2_tag or PA_zz_p_tag.
- virtual const PAlgebra & getZMStar () const =0

  Returns reference to underlying PAlgebra object.
- virtual const vector< ZZX > & getFactorsOverZZ () const =0

  Returns reference to the factorization of Phi_m(X) mod $p^\wedge r$, but as ZZX's.
- virtual long getR () const =0

  The value r.
- virtual long getPPowR () const =0

  The value $p^\wedge r$.
- virtual void restoreContext () const =0

  Restores the NTL context for $p^\wedge r$.
- virtual void genMaskTable () const =0

  Generates the "mask table" that is used to support rotations.

5.26.1 Detailed Description

Virtual base class for PAlgebraMod.

5.26.2 Member Function Documentation

5.26.2.1 virtual void PAlgebraModBase::genMaskTable ( ) const [pure virtual]

Generates the "mask table" that is used to support rotations.

maskTable[i][j] is a polynomial representation of a mask that is 1 in all slots whose i'th coordinate is at least j, and 0 elsewhere. We have:

```
maskTable.size() == zMStar.numOfGens()      // # of generators
for i = 0..maskTable.size()-1:
  maskTable[i].size() == zMStar.OrderOf(i) // order of generator i
```

Implemented in PAlgebraModDerived< type >.

The documentation for this class was generated from the following file:

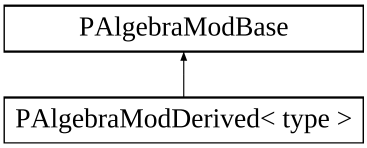
- src/PAlgebra.h

5.27 PAlgebraModDerived< type > Class Template Reference

A concrete instantiation of the virtual class.

```
#include <PAlgebra.h>
```

Inheritance diagram for PAlgebraModDerived< type >:



Public Member Functions

- **PAlgebraModDerived** (const PAlgebra &zMStar, long r)
- **PAlgebraModDerived** (const PAlgebraModDerived &other)
- PAlgebraModDerived & **operator=** (const PAlgebraModDerived &other)
- virtual PAlgebraModBase ∗ clone () const

  Returns a pointer to a "clone".
- virtual PA_tag getTag () const

  Returns the type tag: PA_GF2_tag or PA_zz_p_tag.
- virtual const PAlgebra & getZMStar () const

  Returns reference to underlying PAlgebra object.
- virtual const vector< ZZX > & getFactorsOverZZ () const

  Returns reference to the factorization of Phi_m(X) mod p^r, but as ZZX's.
- virtual long getR () const

  The value r.
- virtual long getPPowR () const

  The value p^r.
- virtual void restoreContext () const

  Restores the NTL context for p^r.
- virtual void genMaskTable () const

  Generates the "mask table" that is used to support rotations.
- const RXModulus & getPhimXMod () const

  Returns reference to an RXModulus representing Phi_m(X) (mod p^r)
- const vec_RX & getFactors () const

  Returns reference to the factors of Phim_m(X) modulo p^r.
- const vec_RX & getCrtCoeffs () const

  Returns the CRT coefficients: element i contains (prod_{j!=i} F_j)^{-1} mod F_i, where F_0 F_1 ... is the factorization of Phi_m(X) mod p^r.
- const vector< vector< RX > > & getMaskTable () const

Returns ref to maskTable, which is used to implement rotations (in the EncryptedArray module).

Embedding in the plaintext slots and decoding back

In all the functions below, G must be irredicible mod p, and the order of G must divide the order of p modulo m (as returned by zMStar.getOrdP()). In addition, when r > 1, G must be the monomial X (RX(1, 1))

- void CRT_decompose (vector< RX > &crt, const RX &H) const
  Returns a vector crt[] such that crt[i] = H mod Ft (with t = T[i])
- void CRT_reconstruct (RX &H, vector< RX > &crt) const
  Returns H in R[X]/Phi_m(X) s.t. for every i<nSlots and t=T[i], we have H == crt[i] (mod Ft)
- void mapToSlots (MappingData< type > &mappingData, const RX &G) const
  Compute the maps for all the slots. In the current implementation, we if r > 1, then we must have either deg(G) == 1 or G == factors[0].
- void embedInAllSlots (RX &H, const RX &alpha, const MappingData< type > &mappingData) const
  Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as alpha in R[X]/G(X).
- void embedInSlots (RX &H, const vector< RX > &alphas, const MappingData< type > &mappingData) const
  Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as alphas[i] in R[X]/G(X).
- void decodePlaintext (vector< RX > &alphas, const RX &ptxt, const MappingData< type > &mapping-Data) const
  Return an array such that alphas[i] in R[X]/G(X) represent the same element as rt = (H mod Ft) in R[X]/Ft(X) where t=T[i].
- void buildLinPolyCoeffs (vector< RX > &C, const vector< RX > &L, const MappingData< type > &mappingData) const
  Returns a coefficient vector C for the linearized polynomial representing M.

5.27.1 Detailed Description

template<class type>class PAlgebraModDerived< type >

A concrete instantiation of the virtual class.

5.27.2 Member Function Documentation

5.27.2.1 template<class type > void PAlgebraModDerived< type >::buildLinPolyCoeffs (vector< RX > & *C*, const vector< RX > & *L*, const MappingData< type > & *mappingData*) const

Returns a coefficient vector C for the linearized polynomial representing M.

For h in Z/(p^r)[X] of degree < d,

$$M(h(X)modG) = sum_{i=0}^{d-1}(C[j]modG)*(h(X^{p^j})modG).$$

G is assumed to be defined in mappingData, with d = deg(G). L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..d-1.

5.27.2.2 template<class type > void PAlgebraModDerived< type >::decodePlaintext (vector< RX > & *alphas*, const RX & *ptxt*, const MappingData< type > & *mappingData*) const

Return an array such that alphas[i] in R[X]/G(X) represent the same element as rt = (H mod Ft) in R[X]/Ft(X) where t=T[i].

The mappingData argument should contain the output of mapToSlots(G).

5.27.2.3 template<class type > void **PAlgebraModDerived**< type >::embedInAllSlots (RX & *H*, const RX & *alpha*, const **MappingData**< type > & *mappingData*) const

Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as alpha in R[X]/G(X).

Must have deg(alpha)<deg(G). The mappingData argument should contain the output of mapToSlots(G).

5.27.2.4 template<class type > void **PAlgebraModDerived**< type >::embedInSlots (RX & *H*, const vector< RX > & *alphas*, const **MappingData**< type > & *mappingData*) const

Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as alphas[i] in R[X]/G(X).

Must have deg(alpha[i])<deg(G). The mappingData argument should contain the output of mapToSlots(G).

5.27.2.5 template<class type > void **PAlgebraModDerived**< type >::genMaskTable ( ) const [virtual]

Generates the "mask table" that is used to support rotations.

maskTable[i][j] is a polynomial representation of a mask that is 1 in all slots whose i'th coordinate is at least j, and 0 elsewhere. We have:

```
maskTable.size() == zMStar.numOfGens()     // # of generators
for i = 0..maskTable.size()-1:
  maskTable[i].size() == zMStar.OrderOf(i) // order of generator i
```

Implements PAlgebraModBase.

5.27.2.6 template<class type> const vector< vector< RX > >& **PAlgebraModDerived**< type >::getMaskTable ( ) const [inline]

Returns ref to maskTable, which is used to implement rotations (in the EncryptedArray module).

maskTable[i][j] is a polynomial representation of a mask that is 1 in all slots whose i'th coordinate is at least j, and 0 elsewhere. We have:

```
maskTable.size() == zMStar.numOfGens()     // # of generators
for i = 0..maskTable.size()-1:
  maskTable[i].size() == zMStar.OrderOf(i) // order of generator i
```

The documentation for this class was generated from the following files:

- src/PAlgebra.h
- src/PAlgebra.cpp

5.28 PlaintextArray Class Reference

A simple wrapper for a pointer to a PlaintextArrayBase. This is the interface that higher-level code should use.

`#include <EncryptedArray.h>`

Public Member Functions

- **PlaintextArray** (const EncryptedArray &ea)
- template<class type >
  const PlaintextArrayDerived
  < type > & **getDerived** (type) const

- template<class type >
  PlaintextArrayDerived< type > & **getDerived** (type)
- const EncryptedArray & getEA () const

  Get the EA object (which is needed for the encoding/decoding routines)
- void rotate (long k)

  Rotation/shift as a linear array.
- void shift (long k)

  Non-cyclic shift with zero fill.
- void encode (const vector< long > &array)

  Encode/decode arrays into plaintext polynomials.
- void **encode** (const vector< ZZX > &array)
- void **decode** (vector< long > &array)
- void **decode** (vector< ZZX > &array)
- void encode (long val)

  Encode with the same value replicated in each slot.
- void **encode** (const ZZX &val)
- void random ()

  Generate a uniformly random element.
- bool equals (const PlaintextArray &other) const

  Equality testing.
- bool **equals** (const vector< long > &other) const
- bool **equals** (const vector< ZZX > &other) const
- void **add** (const PlaintextArray &other)
- void **sub** (const PlaintextArray &other)
- void **mul** (const PlaintextArray &other)
- void **negate** ()
- void replicate (long i)

  Replicate coordinate i at all coordinates.
- void **print** (ostream &s) const

5.28.1 Detailed Description

A simple wrapper for a pointer to a PlaintextArrayBase. This is the interface that higher-level code should use.

The documentation for this class was generated from the following file:

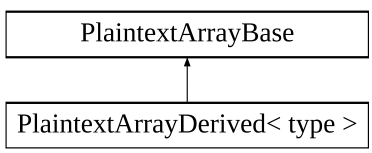
- src/EncryptedArray.h

5.29 PlaintextArrayBase Class Reference

Virtual class for array of slots, not encrypted.

```
#include <EncryptedArray.h>
```

Inheritance diagram for PlaintextArrayBase:



Public Member Functions

- virtual PlaintextArrayBase ∗ **clone** () const =0
- virtual const EncryptedArray & getEA () const =0

  Get the EA object (which is needed for the encoding/decoding routines)
- virtual void rotate (long k)=0

  Rotation/shift as a linear array.
- virtual void shift (long k)=0

  Non-cyclic shift with zero fill.
- virtual void encode (const vector< long > &array)=0

  Encode/decode arrays into plaintext polynomials.
- virtual void **encode** (const vector< ZZX > &array)=0
- virtual void **decode** (vector< long > &array) const =0
- virtual void **decode** (vector< ZZX > &array) const =0
- virtual void encode (long val)=0

  Encode with the same value replicated in each slot.
- virtual void **encode** (const ZZX &val)=0
- virtual void random ()=0

  Generate a uniformly random element.
- virtual bool equals (const PlaintextArrayBase &other) const =0

  Equality testing.
- virtual bool **equals** (const vector< long > &other) const =0
- virtual bool **equals** (const vector< ZZX > &other) const =0
- virtual void **add** (const PlaintextArrayBase &other)=0
- virtual void **sub** (const PlaintextArrayBase &other)=0
- virtual void **mul** (const PlaintextArrayBase &other)=0
- virtual void **negate** ()=0
- virtual void replicate (long i)=0

  Replicate coordinate i at all coordinates.
- virtual void **print** (ostream &s) const =0

5.29.1 Detailed Description

Virtual class for array of slots, not encrypted.

An object pa of type PlaintextArray stores information about an EncryptedArray object ea. The object pa stores a vector of plaintext slots, where each slot is an element of the polynomial ring (Z/(p^r)[X])/(G), where p, r, and G are as defined in ea. Support for arithemetic on PlaintextArray objects is provided.

Mirroring PAlgebraMod and EncryptedArray, we have the following class heirarchy:

PlaintextArrayBase is a virtual class

PlaintextArrayDerived<type> is a derived template class, where type is either PA_GF2 or PA_zz_p.

The class PlaintextArray is a simple wrapper around a smart pointer to a PlaintextArray object: copying a Plaintext-Array object results is a "deep copy" of the underlying object of the derived class.

The documentation for this class was generated from the following file:

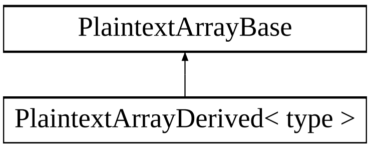
- src/EncryptedArray.h

5.30 PlaintextArrayDerived< type > Class Template Reference

Derived concrete implementation of PlaintextArrayBase.

```
#include <EncryptedArray.h>
```

Inheritance diagram for PlaintextArrayDerived< type >:



Public Member Functions

- virtual PlaintextArrayBase ∗ **clone** () const
- virtual const EncryptedArray & getEA () const

  Get the EA object (which is needed for the encoding/decoding routines)
- **PlaintextArrayDerived** (const EncryptedArray &_ea)
- **PlaintextArrayDerived** (const PlaintextArrayDerived &other)
- PlaintextArrayDerived & **operator=** (const PlaintextArrayDerived &other)
- virtual void rotate (long k)

  Rotation/shift as a linear array.
- virtual void shift (long k)

  Non-cyclic shift with zero fill.
- virtual void encode (const vector< long > &array)

  Encode/decode arrays into plaintext polynomials.
- virtual void **encode** (const vector< ZZX > &array)
- virtual void **decode** (vector< long > &array) const
- virtual void **decode** (vector< ZZX > &array) const
- virtual void encode (long val)

  Encode with the same value replicated in each slot.
- virtual void **encode** (const ZZX &val)
- virtual void random ()

  Generate a uniformly random element.
- virtual bool equals (const PlaintextArrayBase &other) const

  Equality testing.
- virtual bool **equals** (const vector< long > &other) const
- virtual bool **equals** (const vector< ZZX > &other) const
- virtual void **add** (const PlaintextArrayBase &other)
- virtual void **sub** (const PlaintextArrayBase &other)
- virtual void **mul** (const PlaintextArrayBase &other)
- virtual void **negate** ()
- virtual void replicate (long i)

  Replicate coordinate i at all coordinates.
- virtual void **print** (ostream &s) const
- const vector< RX > & **getData** () const
- void **setData** (const vector< RX > &_data)

5.30.1 Detailed Description

template<class type>class PlaintextArrayDerived< type >

Derived concrete implementation of PlaintextArrayBase.

The documentation for this class was generated from the following file:

- src/EncryptedArray.h

5.31 RandomState Class Reference

Facility for "restoring" the NTL PRG state.

`#include <NumbTh.h>`

Public Member Functions

- void restore ()

  Restore the PRG state of NTL.

5.31.1 Detailed Description

Facility for "restoring" the NTL PRG state.

NTL's random number generation faciliity is pretty limited, and does not provide a way to save/restore the state of a pseudo-random stream. This class gives us that ability: Constructing a RandomState object uses the PRG to generate 512 bits and stores them. Upon destruction (or an explicit call to restore()), these bits are used to re-set the seed of the PRG. A typical usage of thie class is as follows:

```
{
  RandomState r;       // save the random state

  SetSeed(something); // set the PRG seed to something
  ...                 // more code that uses the new PRG seed

} // The destructor is called implicitly, PRG state is restored
```

The documentation for this class was generated from the following file:

- src/NumbTh.h

5.32 ReplicateHandler Class Reference

A virtual class to handle call-backs to get the output of replicate.

`#include <replicate.h>`

Public Member Functions

- virtual void **handle** (const Ctxt &ctxt)=0

5.32.1 Detailed Description

A virtual class to handle call-backs to get the output of replicate.

The documentation for this class was generated from the following file:

- src/replicate.h

5.33 shallow_clone< X > Class Template Reference

Shallow copy: initialize with copy constructor.

`#include <cloned_ptr.h>`

Static Public Member Functions

- static X ∗ **apply** (const X ∗x)

5.33.1 Detailed Description

template<class X>class shallow_clone< X >

Shallow copy: initialize with copy constructor.

Template Parameters

X	The class to which this points

The documentation for this class was generated from the following file:

- src/cloned_ptr.h

5.34 SingleCRT Class Reference

This class hold integer polynomials modulo many small primes.

`#include <SingleCRT.h>`

Public Member Functions

- **SingleCRT** (const ZZX &poly, const FHEcontext &_context, const IndexSet &s)
- **SingleCRT** (const ZZX &poly, const FHEcontext &_context)
- **SingleCRT** (const ZZX &poly)
- **SingleCRT** (const FHEcontext &_context, const IndexSet &s)
- **SingleCRT** (const FHEcontext &_context)
- SingleCRT & **operator=** (const SingleCRT &other)
- SingleCRT & **operator=** (const DoubleCRT &dcrt)
- SingleCRT & **operator=** (const ZZX &poly)
- SingleCRT & **operator=** (const ZZ &num)
- SingleCRT & **operator=** (const long num)
- bool **operator==** (const SingleCRT &other) const
- bool **operator!=** (const SingleCRT &other) const
- SingleCRT & **setZero** ()

- SingleCRT & **setOne** ()
- void **addPrimes** (const IndexSet &s1)
- void **removePrimes** (const IndexSet &s1)
- SingleCRT & **operator+=** (const SingleCRT &other)
- SingleCRT & **operator+=** (const ZZX &poly)
- SingleCRT & **operator+=** (const ZZ &num)
- SingleCRT & **operator+=** (long num)
- SingleCRT & **operator-=** (const SingleCRT &other)
- SingleCRT & **operator-=** (const ZZX &poly)
- SingleCRT & **operator-=** (const ZZ &num)
- SingleCRT & **operator-=** (long num)
- void **Add** (const SingleCRT &other, bool matchIndexSet=true)
- void **Sub** (const SingleCRT &other, bool matchIndexSet=true)
- SingleCRT & **operator++** ()
- SingleCRT & **operator--** ()
- void **operator++** (int)
- void **operator--** (int)
- SingleCRT & **operator*=** (const ZZ &num)
- SingleCRT & **operator*=** (long num)
- SingleCRT & **operator/=** (const ZZ &num)
- SingleCRT & **operator/=** (long num)
- void **toPoly** (ZZX &p, const IndexSet &s) const
- void **toPoly** (ZZX &p) const
- const IndexMap< ZZX > & **getMap** () const
- const FHEcontext & **getContext** () const

Friends

- class **DoubleCRT**

5.34.1 Detailed Description

This class hold integer polynomials modulo many small primes.

SingleCRT form is a map from an index set to (ZZX) polynomials, where the i'th polynomial contains the coefficients wrt the ith prime in the chain of moduli. The polynomial thus represented is defined modulo the product of all the primes in use.

This is mostly a helper class for DoubleCRT, and all the rules that apply to DoubleCRT wrt moduli sets apply also to SingleCRT objects. Although SingleCRT and DoubleCRT objects can interact in principle, translation back and forth are expensive since they involve FFT/iFFT. Hence support for interaction between them is limited to explicit conversions.

The documentation for this class was generated from the following files:

- src/SingleCRT.h
- src/SingleCRT.cpp

5.35 SKHandle Class Reference

A handle, describing the secret-key element that "matches" a part, of the form s^r(X^t).

```
#include <Ctxt.h>
```

Public Member Functions

- **SKHandle** (long newPowerOfS=0, long newPowerOfX=1, long newSecretKeyID=0)
- void setBase (long newSecretKeyID=-1)

  Set powerOfS=powerOfX=1.
- bool isBase (long ofKeyID=0) const

  Is powerOfS==powerOfX==1?
- void setOne (long newSecretKeyID=-1)

  Set powerOfS=0, powerOfX=1.
- bool isOne () const

  Is powerOfS==0?
- bool **operator==** (const SKHandle &other) const
- bool **operator!=** (const SKHandle &other) const
- long **getPowerOfS** () const
- long **getPowerOfX** () const
- long **getSecretKeyID** () const
- bool mul (const SKHandle &a, const SKHandle &b)

  Computes the "product" of two handles.

Friends

- class **Ctxt**
- istream & **operator**>> (istream &s, SKHandle &handle)

5.35.1 Detailed Description

A handle, describing the secret-key element that "matches" a part, of the form s^r(X^t).

5.35.2 Member Function Documentation

5.35.2.1 bool SKHandle::mul (const SKHandle & *a*, const SKHandle & *b*) [inline]

Computes the "product" of two handles.

The key-ID's and powers of X must match, else an error state arises, which is represented using a key-ID of -1 and returning false. Also, note that inputs may alias outputs.

To detremine if the resulting handle canbe re-liearized using some key-switchingmatrices from the public key, use the method pubKey.haveKeySWmatrix(handle,handle.secretKeyID), from the class FHEPubKey in FHE.h

The documentation for this class was generated from the following file:

- src/Ctxt.h

Chapter 6

File Documentation

6.1 src/AltCRT.h File Reference

An alternative representation of ring elements.

```
#include <vector>
#include <NTL/ZZX.h>
#include <NTL/lzz_pX.h>
#include "NumbTh.h"
#include "IndexMap.h"
#include "FHEContext.h"
```

Classes

- class AltCRTHelper

  A helper class to enforce consistency within an AltCRT object.
- class AltCRT

  A single-CRT representation of a ring element.

Functions

- void **conv** (AltCRT &d, const ZZX &p)
- AltCRT **to_AltCRT** (const ZZX &p)
- void **conv** (ZZX &p, const AltCRT &d)
- ZZX **to_ZZX** (const AltCRT &d)
- void **conv** (AltCRT &d, const SingleCRT &s)

6.1.1 Detailed Description

An alternative representation of ring elements. The AltCRT module offers a drop-in replacement to DoubleCRT, it exposes the same interface but internally uses a single-CRT representation. That is, polynomials are stored in coefficient representation, modulo each of the small primes in our chain. Currently this class is used only for testing and debugging purposes.

6.2 src/bluestein.h File Reference

declaration of BluesteinFFT(x, a, n, root, powers, Rb):

```
#include <NTL/ZZX.h>
#include <NTL/ZZ_pX.h>
#include <NTL/lzz_pX.h>
```

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)

  bigint implementation
- 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)

  smallint implementation

Variables

- NTL_CLIENT typedef Vec< Vec < mulmod_precon_t > > **fftrep_aux**

6.2.1 Detailed Description

declaration of BluesteinFFT(x, a, n, root, powers, Rb): Compute length-n FFT of the coefficient-vector of x (in place) If the degree of x is less than n then it treats the top coefficients as 0, if the degree of x is more than n then the extra coefficients are ignored. Similarly, if the top entries in x are zeros then x will have degree smaller than n. The argument root is a 2n-th root of unity, namely BluesteinFFT(...,root,...)=DFT(...,root^2,...).

The inverse-FFT is obtained just by calling BluesteinFFT(... root^{-1}), but this procedure is *NOT SCALED*, so BluesteinFFT(x,n,root,...) and then BluesteinFFT(x,n,root^{-1},...) will result in x = n * x_original

In addition, this procedure also returns the powers of root in the powers argument: powers = [1, root, root^4, root^9, ..., root^{(n-1)^2}] and in Rb it returns the size-N FFT representation of the negative powers (with N>=2n-1, N a power of two): b = [0,...,0, root^{-(n-1)^2},...,root^{-4}, root^{-1}, 1, root^{-1},root^{-4},...,root^{-(n-1)^2}, 0...,0] On subsequent calls with these 'powers' and 'Rb', these arrays are not computed again but taken from these pre-comuted variables.

If the powers and Rb arguments are initialized, then it is assumed that they were computed correctly from root. The bahavior is undefined when calling with initialized powers and Rb but a different root. In particular, to compute the inverse-FFT (using root^{-1}), one must provide different powers and Rb arguments than those that were given when computing in the forward direction using root. To reset these arguments between calls with different root values, use clear(powers); Rb.SetSize(0);

Ra is just a scratch FFT rep, supplied by the caller to minimize memory allocations.

This module builds on Shoup's NTL, and contains both a bigint version with types ZZ_p and ZZ_pX and a smallint version with types zz_p and zz_pX.

6.3 src/cloned_ptr.h File Reference

Implemenation of smart pointers with "deep cloning" semantics.

Classes

- class deep_clone< X >

  Deep copy: initialize with clone.
- class shallow_clone< X >

Shallow copy: initialize with copy constructor.

Macros

- #define **CLONED_PTR_TEMPLATE_MEMBERS**(CLONED_PTR_TYPE)
- #define **CLONED_PTR_DECLARE**(CLONED_PTR_TYPE, CLONED_PTR_INIT)

6.3.1 Detailed Description

Implemenation of smart pointers with "deep cloning" semantics. Based (loosely) on code from

http://github.com/yonat/smart_ptr/blob/master/cloned_ptr.h

6.3.2 Macro Definition Documentation

6.3.2.1 #define CLONED_PTR_TEMPLATE_MEMBERS(*CLONED_PTR_TYPE*)

Value:

```
\
    template <class Y> CLONED_PTR_TYPE(const CLONED_PTR_TYPE<Y>& r) \
        {copy(r.ptr);} \
    template <class Y> CLONED_PTR_TYPE& operator=(const CLONED_PTR_TYPE<Y>& r) \
    { \
        if (this != &r) { \
            delete ptr; \
            copy(r.ptr); \
        } \
        return *this; \
    } \
```

6.4 src/CModulus.h File Reference

Supports forward and backward length-m FFT transformations.

```
#include "PAlgebra.h"
#include "bluestein.h"
#include "cloned_ptr.h"
```

Classes

- class CMOD_zz_p

  typedefs for smallint Cmodulus
- class CMOD_ZZ_p

  typedefs for bigint CModulus
- class Cmod< type >

  template class for both bigint and smallint implementations

Macros

- #define **INJECT_TYPE**(type, subtype) typedef typename type::subtype subtype

Typedefs

- typedef Cmod< CMOD_zz_p > **Cmodulus**
- typedef Cmod< CMOD_ZZ_p > **CModulus**

6.4.1 Detailed Description

Supports forward and backward length-m FFT transformations. This is a wrapper around the bluesteinFFT routines, for one modulus q. Two classes are defined here, Cmodulus for a small moduli (long) and CModulus for a large ones (ZZ). These classes are otherwise identical hence they are implemented using a class template.

6.5 src/Ctxt.h File Reference

Declerations of a BGV-type cipehrtext and key-switching matrices.

```
#include <vector>
#include <NTL/xdouble.h>
#include "DoubleCRT.h"
```

Classes

- class SKHandle

  A handle, describing the secret-key element that "matches" a part, of the form s^r(X^t).
- class CtxtPart

  One entry in a ciphertext vector.
- class Ctxt

  A Ctxt object holds a single cipehrtext.

Functions

- ostream & **operator<<** (ostream &s, const SKHandle &handle)
- istream & **operator>>** (istream &s, CtxtPart &p)
- ostream & **operator<<** (ostream &s, const CtxtPart &p)
- IndexSet **baseSetOf** (const Ctxt &c)
- void **CheckCtxt** (const Ctxt &c, const char *label)

6.5.1 Detailed Description

Declerations of a BGV-type cipehrtext and key-switching matrices. A ciphertext is a vector of "ciphertext parts", each part consists of a polynomial (element of polynomial ring R_Q) and a "handle" describing the secret-key polynomial that this part multiplies during decryption. For example:

- A "canonical" ciphertext has two parts, the first part multiplies 1 and the second multiplies the "base" secret key s.
- When you multiply two canonical ciphertexts you get a 3-part ciphertext, with parts corresponding to 1, s, and s^2.
- When you apply automorphism X->X^t to a generic ciphertext, then
  - the part corresponding to 1 still remains wrt 1

- every other part corresponding to some s' will now be corresponding to the polynomial s'(X^t) mod Phi_m(X)

This type of representation lets you in principle add ciphertexts that are defined with respect to different keys:

- For parts of the two cipehrtexts that point to the same secret-key polynomial, you just add the two Double-CRT polynomials
- Parts in one ciphertext that do not have counter-part in the other cipehrtext will just be included in the result intact. For example, you have the ciphertexts C1 = (a relative to 1, b relative to s) C2 = (u relative to 1, v relative to s(X^3)) Then their sum will be C1+C2 = (a+u relative to 1, b relative to s, v relative to s(X^3))

Similarly, in principle you can also multiply arbitrary cipehrtexts, even ones that are defined with respect to different keys, and the result will be defined with respect to the tensor product of the two keys.

The current implementation is more restrictive, however. It requires that a ciphertext has one part wrt 1, that for every r>=1 there is at most one part wrt to s^r(X^t) (for some t), and that the r's are consecutive. For example you cannot have parts wrt (1,s,s^3) without having a part wrt s^2.

It follows that you can only add/multiply ciphertexts if one of the two lists of handles is a prefix of the other. For example, one can add a ciphertext wrt (1,s(X^2)) to another wrt (1,s(X^2),s^2(X^2)), but not to another ciphertext wrt (1,s).

6.6 src/DoubleCRT.h File Reference

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

```
#include <vector>
#include <NTL/ZZX.h>
#include <NTL/vec_vec_long.h>
#include "NumbTh.h"
#include "IndexMap.h"
#include "FHEContext.h"
```

Classes

- class DoubleCRTHelper

    A helper class to enforce consistency within an DoubleCRTHelper object.
- class DoubleCRT

    Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

Functions

- void **conv** (DoubleCRT &d, const ZZX &p)
- DoubleCRT **to_DoubleCRT** (const ZZX &p)
- void **conv** (ZZX &p, const DoubleCRT &d)
- ZZX **to_ZZX** (const DoubleCRT &d)
- void **conv** (DoubleCRT &d, const SingleCRT &s)

6.6.1 Detailed Description

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

6.7 src/EncryptedArray.h File Reference

Data-movement operations on encrypted arrays of slots.

```
#include "FHE.h"
#include <NTL/ZZ_pX.h>
#include <NTL/GF2X.h>
#include <NTL/ZZX.h>
```

Classes

- class EncryptedArrayBase

  virtual class for data-movement operations on arrays of slots
- class EncryptedArrayDerived< type >

  Derived concrete implementation of EncryptedArrayBase.
- class EncryptedArray

  A simple wrapper for a smart pointer to an EncryptedArrayBase. This is the interface that higher-level code should use.
- class PlaintextArrayBase

  Virtual class for array of slots, not encrypted.
- class PlaintextArrayDerived< type >

  Derived concrete implementation of PlaintextArrayBase.
- class PlaintextArray

  A simple wrapper for a pointer to a PlaintextArrayBase. This is the interface that higher-level code should use.

Functions

- EncryptedArrayBase ∗ buildEncryptedArray (const FHEcontext &context, const ZZX &G)

  A "factory" for building EncryptedArrays.
- PlaintextArrayBase ∗ buildPlaintextArray (const EncryptedArray &ea)

  A "factory" for building EncryptedArrays.

6.7.1 Detailed Description

Data-movement operations on encrypted arrays of slots.

6.8 src/FHE.h File Reference

Public/secret keys for the BGV cryptosystem.

```
#include <vector>
#include "NTL/ZZX.h"
#include "DoubleCRT.h"
#include "FHEContext.h"
#include "Ctxt.h"
```

Classes

- class KeySwitch

  Key-switching matrices.
- class FHEPubKey

  The public key.
- class FHESecKey

  The secret key.

Functions

- ostream & **operator**<< (ostream &str, const KeySwitch &matrix)

Strategies for generating key-switching matrices

These functions are implemented in KeySwitching.cpp

- void addAllMatrices (FHESecKey &sKey, long keyID=0)

  Maximalistic approach: generate matrices s(X^e)->s(X) for all e in Zm∗.
- void addFewMatrices (FHESecKey &sKey, long keyID=0)

  Generate matrices so every s(X^e) can be reLinearized in at most two steps.
- void add1DMatrices (FHESecKey &sKey, long keyID=0)

  Generate all matrices s(X^{g^i})->s(X) for generators g of Zm∗ /(p) and i<ord(g). If g has different orders in Zm∗ and Zm∗ /(p) then generate also matrices of the form s(X^{g^{-i}})->s(X)
- void addSome1DMatrices (FHESecKey &sKey, long bound=100, long keyID=0)

  Generate some matrices of the form s(X^{g^i})->s(X), but not all. For a generator g whose order is larger than bound, generate only enough matrices for the giant-step/baby-step procedures (2∗sqrt(ord(g))of them).
- void addFrbMatrices (FHESecKey &sKey, long keyID=0)

  Generate all Frobenius matrices of the form s(X^{2^i})->s(X)

6.8.1 Detailed Description

Public/secret keys for the BGV cryptosystem.

6.9 src/FHEContext.h File Reference

Keeps the parameters of an instance of the cryptosystem.

```
#include <NTL/xdouble.h>
#include "PAlgebra.h"
#include "CModulus.h"
#include "IndexSet.h"
```

Classes

- class FHEcontext

  Maintaining the parameters.

Functions

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

    Returns smallest parameter m satisfying various constraints:
- void writeContextBase (ostream &s, const FHEcontext &context)

    write [m p r] data
- void readContextBase (istream &s, unsigned &m, unsigned &p, unsigned &r)

    read [m p r] data, needed to construct context

Convenience routines for generating the modulus chain

- double AddPrimesBySize (FHEcontext &context, double totalSize, bool special=false)

    Adds to the chain primes whose product is at least e^totalSize, returns natural log of the product of all added primes.
- double AddPrimesByNumber (FHEcontext &context, long nPrimes, long startAt=1, bool special=false)

    Adds n primes to the chain returns natural log of the product of all added primes.
- void buildModChain (FHEcontext &context, long nLvls, long c=3)

    Build modulus chain for nLvls levels, using c digits in key-switching.

Variables

- FHEcontext ∗ **activeContext**

6.9.1 Detailed Description

Keeps the parameters of an instance of the cryptosystem.

6.9.2 Function Documentation

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

Returns smallest parameter m satisfying various constraints:

Parameters

k	security parameter
L	number of levels
c	number of columns in key switching matrices
p	characteristic of plaintext space
d	embedding degree (d ==0 or d==1 => no constraint)
s	at least that many plaintext slots
chosen_m	preselected value of m (0 => not preselected) Fails with an error message if no suitable m is found prints an informative message if verbose == true

6.10 src/IndexMap.h File Reference

Implementation of a map indexed by a dynamic set of integers.

```
#include "IndexSet.h"
#include <tr1/unordered_map>
#include <iostream>
#include <cassert>
#include "cloned_ptr.h"
```

Classes

- class IndexMapInit< T >

  Initializing elements in an IndexMap.
- class IndexMap< T >

  IndexMap<T> implements a generic map indexed by a dynamic index set.

Functions

- template< class T >
  bool operator== (const IndexMap< T > &map1, const IndexMap< T > &map2)

  Comparing maps, by comparing all the elements.
- template< class T >
  bool **operator!=** (const IndexMap< T > &map1, const IndexMap< T > &map2)

6.10.1 Detailed Description

Implementation of a map indexed by a dynamic set of integers.

6.11 src/IndexSet.h File Reference

A dynamic set of integers.

```
#include <vector>
#include <iostream>
#include <cassert>
```

Classes

- class IndexSet

  A dynamic set of non-negative integers.

Functions

- IndexSet operator| (const IndexSet &s, const IndexSet &t)

  union
- IndexSet operator& (const IndexSet &s, const IndexSet &t)

  intersection
- IndexSet operator^ (const IndexSet &s, const IndexSet &t)

  exclusive-or
- IndexSet operator/ (const IndexSet &s, const IndexSet &t)

  set minus
- ostream & **operator**<< (ostream &str, const IndexSet &set)
- istream & **operator**>> (istream &str, IndexSet &set)
- long card (const IndexSet &s)

  Functional cardinality.
- bool **empty** (const IndexSet &s)

- bool operator<= (const IndexSet &s1, const IndexSet &s2)

  Is s1 subset or equal to s2.
- bool operator< (const IndexSet &s1, const IndexSet &s2)

  Is s1 strict subset of s2.
- bool operator>= (const IndexSet &s1, const IndexSet &s2)

  Is s2 subset or equal to s2.
- bool operator> (const IndexSet &s1, const IndexSet &s2)

  Is s2 strict subset of s1.
- bool disjoint (const IndexSet &s1, const IndexSet &s2)

  Functional disjoint.

6.11.1 Detailed Description

A dynamic set of integers.

6.12 src/NumbTh.h File Reference

Miscellaneous utility functions.

```
#include <vector>
#include <cmath>
#include <istream>
#include <NTL/ZZ.h>
#include <NTL/ZZ_p.h>
#include <NTL/ZZX.h>
#include <NTL/GF2X.h>
#include <NTL/vec_ZZ.h>
#include <NTL/xdouble.h>
#include <NTL/mat_lzz_pE.h>
#include <NTL/mat_GF2E.h>
#include <NTL/lzz_pXFactoring.h>
#include <NTL/GF2XFactoring.h>
#include <tr1/unordered_map>
#include <string>
```

Classes

- class RandomState

  Facility for "restoring" the NTL PRG state.

Typedefs

- typedef tr1::unordered_map
  < string, const char * > **argmap_t**

Functions

- bool parseArgs (int argc, char *argv[], argmap_t &argmap)

  Code for parsing command line arguments.
- long multOrd (long p, long m)

  Return multiplicative order of p modulo m, or 0 if GCD(p, m) != 1.

- void ppsolve (vec_zz_pE &x, const mat_zz_pE &A, const vec_zz_pE &b, long p, long r)

  Prime power solver.
- void ppsolve (vec_GF2E &x, const mat_GF2E &A, const vec_GF2E &b, long p, long r)

  A version for GF2: must have p == 2 and r == 1.
- void buildLinPolyCoeffs (vec_zz_pE &C, const vec_zz_pE &L, long p, long r)

  Combination of buildLinPolyMatrix and ppsolve.
- void buildLinPolyCoeffs (vec_GF2E &C, const vec_GF2E &L, long p, long r)

  A version for GF2: must be called with p == 2 and r == 1.
- void applyLinPoly (zz_pE &beta, const vec_zz_pE &C, const zz_pE &alpha, long p)

  Apply a linearized polynomial with coefficient vector C.
- void applyLinPoly (GF2E &beta, const vec_GF2E &C, const GF2E &alpha, long p)

  A version for GF2: must be called with p == 2 and r == 1.
- double log2 (const xdouble &x)

  Base-2 logarithm.
- double **log2** (const double x)
- void factorize (vector< long > &factors, long N)

  Factoring by trial division, only works for N<2^{60}, only the primes are recorded, not their multiplicity.
- void **factorize** (vector< ZZ > &factors, const ZZ &N)
- void phiN (long &phiN, vector< long > &facts, long N)

  Compute Phi(N) and also factorize N.
- void **phiN** (ZZ &phiN, vector< ZZ > &facts, const ZZ &N)
- int phi_N (int N)

  Compute Phi(N).
- void FindPrimitiveRoot (zz_p &r, unsigned e)

  Find e-th root of unity modulo the current modulus.
- void **FindPrimitiveRoot** (ZZ_p &r, unsigned e)
- int mobius (int n)

  Compute mobius function (naive method as n is small).
- ZZX Cyclotomic (int N)

  Compute cyclotomic polynomial.
- int primroot (int N, int phiN)

  Find a primitive root modulo N.
- int ord (int N, int p)

  Compute the highest power of p that divides N.
- ZZX **RandPoly** (int n, const ZZ &p)
- void MulMod (ZZX &out, const ZZX &f, long a, long q, bool abs=true)

  Multiply the polynomial f by the integer a modulo q.
- ZZX **MulMod** (const ZZX &f, long a, long q, bool abs=true)
- template<class T1 , class T2 >
  void convert (T1 &x1, const T2 &x2)

  A generic template that resolves to NTL's conv routine.
- template<class T1 , class T2 >
  void convert (vector< T1 > &v1, const vector< T2 > &v2)

  A generic vector conversion routine.
- void **mul** (vector< ZZX > &x, const vector< ZZX > &a, long b)
- void **div** (vector< ZZX > &x, const vector< ZZX > &a, long b)
- void **add** (vector< ZZX > &x, const vector< ZZX > &a, const vector< ZZX > &b)
- int is_in (int x, int *X, int sz)

  Finds whether x is an element of the set X of size sz, Returns -1 it not and the location if true.
- template<class zzvec >
  bool intVecCRT (vec_ZZ &vp, const ZZ &p, const zzvec &vq, long q)

  Incremental integer CRT for vectors.

- template<class T , bool maxFlag>
  long argminmax (vector< T > &v)

  Find the index of the (first) largest/smallest element.
- template<class T >
  long **argmax** (vector< T > &v)
- template<class T >
  long **argmin** (vector< T > &v)
- void sampleSmall (ZZX &poly, long n=0)

  Sample polynomials with entries {-1,0,1}. Each coefficient is 0 with probability 1/2 and +-1 with probability 1/4.
- void sampleHWt (ZZX &poly, long Hwt, long n=0)

  Sample polynomials with entries {-1,0,1} with a given HAming weight.
- void sampleGaussian (ZZX &poly, long n=0, double stdev=1.0)

  Sample polynomials with Gaussian coefficients.
- void seekPastChar (istream &str, int cc)

  Advance the input stream beyond white spaces and a single instance of the char cc.
- template<typename T >
  long lsize (const vector< T > &v)

  Size of STL vector as a long (rather than unsigned long)
- template<typename T1 , typename T2 >
  bool sameObject (const T1 ∗p1, const T2 ∗p2)

  Testing if two vectors point to the same object.
- void ModComp (ZZX &res, const ZZX &g, const ZZX &h, const ZZX &f)

  Modular composition of polynomials: res = g(h) mod f.
- void PolyRed (ZZX &out, const ZZX &in, int q, bool abs=false)

  Reduce all the coefficients of a polynomial modulo q.
- void **PolyRed** (ZZX &out, const ZZX &in, const ZZ &q, bool abs=false)
- void **PolyRed** (ZZX &F, int q, bool abs=false)
- void **PolyRed** (ZZX &F, const ZZ &q, bool abs=false)

Some enhanced conversion routines

- void **convert** (long &x1, const GF2X &x2)
- void **convert** (long &x1, const zz_pX &x2)
- void **convert** (vec_zz_pE &X, const vector< ZZX > &A)
- void **convert** (mat_zz_pE &X, const vector< vector< ZZX > > &A)
- void **convert** (vector< ZZX > &X, const vec_zz_pE &A)
- void **convert** (vector< vector< ZZX > > &X, const mat_zz_pE &A)

The size of the coefficient vector of a polynomial.

- ZZ **sumOfCoeffs** (const ZZX &f)
- ZZ **largestCoeff** (const ZZX &f)
- xdouble **coeffsL2Norm** (const ZZX &f)

6.12.1 Detailed Description

Miscellaneous utility functions.

6.12.2 Function Documentation

6.12.2.1 void applyLinPoly (zz_pE & *beta,* const vec_zz_pE & *C,* const zz_pE & *alpha,* long *p*)

Apply a linearized polynomial with coefficient vector C.

NTL's current smallint modulus, zz_p::modulus(), is assumed to be p^r, for p prime, r >= 1 integer.

6.12.2.2 template<class T , bool maxFlag> long argminmax (vector< T > & *v*)

Find the index of the (first) largest/smallest element.

These procedures are roughly just simpler variants of std::max_element and std::min_element. argmin/argmax are implemented as a template, so the code must be placed in the header file for the comiler to find it. The class T must have an implementation of operator> and operator< for this template to work.

Template Parameters

maxFlag	A boolean value: true - argmax, false - argmin

6.12.2.3 void buildLinPolyCoeffs (vec_zz_pE & *C*, const vec_zz_pE & *L*, long *p*, long *r*)

Combination of buildLinPolyMatrix and ppsolve.

Obtain the linearized polynomial coefficients from a vector L representing the action of a linear map on the standard basis for zz_pE over zz_p.

NTL's current smallint modulus, zz_p::modulus(), is assumed to be p^r, for p prime, r >= 1 integer.

6.12.2.4 template<class zzvec > bool intVecCRT (vec_ZZ & *vp*, const ZZ & *p*, const zzvec & *vq*, long *q*)

Incremental integer CRT for vectors.

Expects co-primes p,q with q odd, and such that all the entries in v1 are in [-p/2,p/2). Returns in v1 the CRT of vp mod p and vq mod q, as integers in [-pq/2, pq/2). Uses the formula:

$$CRT(vp,p,vq,q) = vp + [(vq - vp) * p^{-1}]_q * p,$$

where [...]_q means reduction to the interval [-q/2,q/2). Notice that if q is odd then this is the same as reducing to [-(q-1)/2,(q-1)/2], which means that [...]_q * p is in [-p(q-1)/2, p(q-1)/2], and since vp is in [-p/2,p/2) then the sum is indeed in [-pq/2,pq/2).

Return true is both vectors are of the same length, false otherwise

6.12.2.5 bool parseArgs (int *argc*, char * *argv[ ]*, argmap_t & *argmap*)

Code for parsing command line arguments.

Tries to parse each argument as arg=val, and returns a correspinding map. It returns false if errors were detected, and true otherwise.

6.12.2.6 void PolyRed (ZZX & *out*, const ZZX & *in*, int *q*, bool *abs* = `false`)

Reduce all the coefficients of a polynomial modulo q.

When abs=false reduce to interval (-q/2,...,q/2), when abs=true reduce to [0,q). When abs=false and q=2, maintains the same sign as the input.

6.12.2.7 void ppsolve (vec_zz_pE & *x*, const mat_zz_pE & *A*, const vec_zz_pE & *b*, long *p*, long *r*)

Prime power solver.

A is an n x n matrix, b is a length n (row) vector, this function finds a solution for the matrix-vector equation x A = b. An error is raised if A is not inverible mod p.

NTL's current smallint modulus, zz_p::modulus(), is assumed to be p^r, for p prime, r >= 1 integer.

6.12.2.8 void sampleHWt (ZZX & *poly*, long *Hwt*, long *n* = 0)

Sample polynomials with entries {-1,0,1} with a given HAming weight.

Choose min(Hwt,n) coefficients at random in {-1,+1} and the others are set to zero. If n=0 then n=poly.deg()+1 is used.

6.13 src/PAlgebra.h File Reference

Declatations of the classes PAlgebra.

```
#include <vector>
#include <NTL/ZZX.h>
#include <NTL/GF2X.h>
#include <NTL/vec_GF2.h>
#include <NTL/GF2EX.h>
#include <NTL/lzz_pEX.h>
#include "cloned_ptr.h"
```

Classes

- class PAlgebra

  The structure of (Z/mZ)∗ /(p)
- class PAlgebraModBase

  Virtual base class for PAlgebraMod.
- class PAlgebraModDerived< type >

  A concrete instantiation of the virtual class.
- class MappingData< type >

  Auxilliary structure to support encoding/decoding slots.
- class PAlgebraModDerived< type >

  A concrete instantiation of the virtual class.
- class PAlgebraMod

  The structure of Z[X]/(Phi_m(X), p)

Enumerations

- enum **PA_tag** { **PA_GF2_tag**, **PA_zz_p_tag** }

Functions

- PAlgebraModBase ∗ buildPAlgebraMod (const PAlgebra &zMStar, long r)

  Builds a table, of type PA_GF2 if p == 2 and r == 1, and PA_zz_p otherwise.

6.13.1 Detailed Description

Declatations of the classes PAlgebra.

6.14 src/replicate.h File Reference

Procedures for replicating a ciphertext slot across a full ciphertext.

```
#include "FHE.h"
#include "EncryptedArray.h"
```

Classes

- class ReplicateHandler

  A virtual class to handle call-backs to get the output of replicate.

Functions

- void replicate (const EncryptedArray &ea, Ctxt &ctx, long pos)

  The value in slot #pos is replicated in all other slots. On an n-slot ciphertext, this algorithm performs O(log n) 1D rotations.
- void replicate0 (const EncryptedArray &ea, Ctxt &ctxt, long pos)

  A lower-level routine. Same as replicate, but assumes all slots are zero except slot #pos.
- void replicateAll (const EncryptedArray &ea, const Ctxt &ctxt, ReplicateHandler ∗handler, long recBound=64)
- void replicateAllOrig (const EncryptedArray &ea, const Ctxt &ctxt, ReplicateHandler ∗handler)

Variables

- bool **replicateVerboseFlag**

6.14.1 Detailed Description

Procedures for replicating a ciphertext slot across a full ciphertext. This module implements a recursive, O(1)-amortized algorithm for replications. On an input ciphertext that encrypts (x_1, ..., x_n), we generate the n encrypted vectors (x_1, ..., x_1), ..., (x_n, ..., x_n), in that order.

To process the output vectors, a "call back" mechanism is used (so that we do'ne need to generate them all, and instead can return them one by one). For this purpose, the caller should pass a pointer to a class derived from the purely abstract class ReplicateHandler.

The replication procedures are meant to be used for linear algebra operation where a matrix-vector multiplication can be implemented for example by replicating each entry of the vector as a stand-alone ciphertext, then use the SIMD operations on these ciphertexts.

6.14.2 Function Documentation

6.14.2.1 void replicateAll (const **EncryptedArray** & *ea,* const **Ctxt** & *ctxt,* **ReplicateHandler** ∗ *handler,* long *recBound =* `64`)

replicateAll uses a hybrid strategy, combining the O(log n) strategy of the replicate method, with an O(1) strategy, which is faster but introduces more noise. This tradeoff is controlled by the parameter recBound:

- recBound $<$ 0: recursion to depth $|$recBound$|$ (faster, noisier)
- recBound ==0: no recursion (slower, less noise)
- recBound $>$ 0: the recursion depth is chosen heuristically, but is capped at recBound

The default value for recBound is 64, this ensures that the choice is based only on the heuristic, which will introduce noise corresponding to O(log log n) levels of recursion, but still gives an algorithm that theoretically runs in time O(n).

6.14.2.2 void replicateAllOrig (const **EncryptedArray** & *ea*, const **Ctxt** & *ctxt*, **ReplicateHandler** ∗ *handler*)

This function is obsolete, and is kept for historical purposes only. It was a first attempt at implementing the O(1)-amortized algorithm, but is less efficient than the function above.

6.15 src/SingleCRT.h File Reference

Decleration for the helper SingleCRT class.

```
#include <vector>
#include <iostream>
#include <NTL/ZZX.h>
#include "FHEContext.h"
#include "IndexMap.h"
#include "DoubleCRT.h"
```

Classes

- class SingleCRT

  This class hold integer polynomials modulo many small primes.

Functions

- void **conv** (SingleCRT &s, const ZZX &p)
- void **conv** (ZZX &p, const SingleCRT &s)
- ZZX **to_ZZX** (const SingleCRT &s)
- void **conv** (SingleCRT &s, const DoubleCRT &d)

6.15.1 Detailed Description

Decleration for the helper SingleCRT class.

6.16 src/timing.h File Reference

Utility functions for measuering time.

```
#include <iostream>
```

Macros

- #define **FHE_TIMER_START** {if (areTimersOn()) startFHEtimer(__func__);}
- #define **FHE_TIMER_STOP** {if (areTimersOn()) stopFHEtimer(__func__);}
- #define **FHE_NTIMER_START**(n) {if (areTimersOn()) startFHEtimer(n);}
- #define **FHE_NTIMER_STOP**(n) {if (areTimersOn()) stopFHEtimer(n);}

Functions

- void **setTimersOn** ()
- void **setTimersOff** ()
- bool **areTimersOn** ()
- void startFHEtimer (const char ∗fncName)

  Start a timer.
- void stopFHEtimer (const char ∗fncName)

  Stop a timer.
- void resetFHEtimer (const char ∗fncName)

  Reset a timer for some label to zero.
- double getTime4func (const char ∗fncName)

  Read the value of a timer (in seconds)
- long getNumCalls4func (const char ∗fncName)

  Returns number of calls for that timer.
- void **resetAllTimers** ()
- void printAllTimers (std::ostream &str=std::cerr)

  Print the value of all timers to stream.

Variables

- bool **FHEtimersOn**

6.16.1 Detailed Description

Utility functions for measuering time. 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 setTimers-On() to activate the timers and setTimersOff() to pause them. To obtain the value of a given timer (in seconds), the application can use the function getTime4func(const char ∗fncName), and the function printAllTimers() prints the values of all timers to an output stream.

Using this method 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__). We can also use the "lower level" methods startFH-Etimer(name), stopFHEtimer(name), and resetFHEtimer(name) to add timers with arbitrary names (not necessarily associated with functions).

Index