PALISADE Lattice Cryptography Library
User Manual (v1.10.4)

Yuriy Polyakov[1,2], Kurt Rohloff[1,2], Gerard W. Ryan[2], and Dave Cousins[1,2]

[1]Duality Technologies, Newark, NJ, 07102, USA.
{ypolyakov,krohloff,dcousins}@dualitytech.com
[2]Cybersecurity Research Center, New Jersey Institute of Technology (NJIT),
Newark, NJ, 07102, USA. {polyakov,rohloff,dcousins}@njit.edu

September 18, 2020

Abstract

This document is the manual for the PALISADE lattice cryptography library. This manual provides an introduction to the library by describing the library architecture and cataloging its capabilities. We focus on the PALISADE library's ability to support homomorphic encryption capabilities to evaluate arithmetic operations on data while encrypted. We do not explicitly provide an introduction to lattice cryptography, but we provide an overview of notation and terminology necessary to use the PALISADE library. In addition to providing code samples for the use of the PALISADE library, we also discuss the library programming style for developers who wish to read library code or even add to the library. We also provide an overview of common pitfalls in the use of PALISADE.

Contents

Listings

1 Document Overview

This manuscript is a working document to introduce users to the PALISADE lattice cryptography library. The most recent copy of this document is available for download from the PALISADE library website. Copies of this document may be available for download from PALISADE contributors' websites, but the version available for download on the PALISADE library website should be considered authoritative.

The manual is organized as follows;

- An introduction to the library is provided in Section 2.
- Section 3 provides a basic introduction to lattice cryptography.
- The library architecture is discussed in Section 4.
- The library capabilities are discussed in Section 5.
- Section 6 provides an overview of the PALISADE library directory structure.
- Section 7 provides an overview of notation and terminology used in the library and this manual.
- Section 8 provides codes samples for the Public Key Encryption (`PKE`) module.
- Section 9 provides codes samples covering FHE for integer arithmetic, real-number arithmetic, and Boolean circuits.
- Section 10 provides codes samples for Attribute-based Encryption (`ABE`).
- Section 11 provides codes samples for Digital Signatures (`Signature`).
- Section 12 describes how to build the library in a generic Linux environment.
- Section 13 describes the programming style we maintain in the library for those users who wish to contribute code to the library.

In addition, we provide the following appendices:

- Appendix A reproduces the library's BSD 2-clause license.
- The PALISADE library has been made possible by the generous support of our sponsors. A listing of the PALISADE library sponsors can been seen in Appendix B.

The public version of the PALISADE Lattice Cryptography library can be found on The PALISADE Git Repository.

The listed authors of this document are the current primary maintainers of the PALISADE library repository.

2 Introduction

Lattice cryptography has received considerable attention because of its capability to support both post-quantum public-key encryption and the ability to compute on data while encrypted via homomorphic encryption. Lattice cryptography also provides other powerful capabilities such as proxy re-encryption and attribute-based encryption. The PALISADE library provides implementations of the building blocks for lattice cryptography capabilities along with end-to-end implementations of advanced lattice cryptography protocols for public-key encryption, proxy re-encryption, homomorphic encryption and others. PALISADE provides both an experimental platform for researchers to design and evaluate new lattice cryptography capabilities, while at the same time providing implementations of known protocols that can be integrated into applications. In this manual we describe PALISADE and discuss how it can be used.

The PALISADE library is designed to address the following inherent challenges of lattice cryptography implementation:

- The complexity of algebraic constructions makes it hard for non-experts to leverage, let alone implement, algebraic constructions used as the building blocks of lattice cryptography.
- Lattice cryptography implementations are often purpose-built. Rapid deployment on new hardware systems is difficult to support with these existing implementations.
- Parameter selection for security and performance takes very sophisticated understanding, and this is nontrivial even for experts - up to a dozen parameters may be needed to be set for some schemes.
- Security assumptions are evolving, and it has been difficult to adapt prior implementations. See for example the recent sub-field lattice attacks against LTV which required re-design of libraries that previously used this scheme.
- Application integration has been challenging, without easy methods to efficiently perform operations on non-trivial data types, such as rationals, complex numbers, etc.

As a result of these identified challenges, we design PALISADE to achieve the following design goals:

- *Create an extensible and adaptable library for lattice cryptography.* Lattice crypto uses several computational primitives. We allow for new protocols that mix-and-match these primitives to avoid the need for expert low-level knowledge. This has been a major advance in the PALISADE library, and we are still refining these features of PALISADE as we support more capabilities.
- *Provide a modular structure to mix and match components.* This allows for system-optimized arithmetic and lattice "back-ends"/plugins. We currently support multiple implementations of big integer math that have various performance trade offs and bit length limitations. Multiple sets of math back-ends can be selected at compile

time. The PALISADE library is thus designed to be highly portable into commodity computing and hardware environments, including Windows, Linux, MacOS and Android environments (though Android it experimental). The ability to support multiple hardware accelerators is a work in progress in PALISADE.

- *Offer (semi-)automated parameter selection.* This reduces the need to tune an overwhelming number of parameters. This is still a major research topic and is a work in progress in PALISADE.
- *Develop common crypto APIs.* Common crypto APIs across multiple schemes and back-ends "hide" complex details of lattice constructions / parameterization from application developers, allowing developers to focus on their areas of interest while integrating and replacing components to maintain security and performance. This is another major feature of PALISADE that we continue to refine.
- *Deliver good software engineering with focus on usability.* This permits standards-based design and style. Unit tests and bench-marking environments are supported for evaluation and tuning of integrated applications. We aim to provide documentation, clean code and useful examples that reduce effort for new developers.

Our identification of these goals in PALISADE is informed by industry experience integrating PALISADE into multi-organizational large software engineering projects. As a result of these identified challenges and the resulting engineering goals we set for ourselves in the design of PALISADE, we designed PALISADE to be highly modular, with a core library of lattice cryptography primitives that support multiple protocols for public-key encryption, homomorphic encryption, digital signature schemes, proxy re-encryption and program obfuscation.

3 A Brief Overview of Lattice Cryptography

In the following discussion, objects and functions that are directly supported by PALISADE are printed in `this` type.

At a high intuitive level, encryption is a computational process wherein `Data` is `Encoded` as `Plaintext` and then encrypted into `Ciphertext` according to some `Encryption` algorithm. Conversely, `Decryption` is a corresponding computational process wherein the `Plaintext` can be recovered from the `Ciphertext` through the use of a corresponding `Decryption` algorithm.

These algorithms use cryptographic `Keys` to perform the `Encryption` and `Decryption` operations. Intuitively, an encryption scheme is secure if it prevents an adversary from recovering `Plaintext` (or information about a `Plaintext`) from `Ciphertext` when the adversary does not have the corresponding decryption key.

A "symmetric key" protocol uses the same key to perform both encryption and decryption. A "public key" or "assymmetric" protocol uses a pair of `Public` and `Secret keys`, respectively, for encryption and decryption. A `Secret key` is sometimes called a `Private key` .

In practice symmetric keys and secret keys are kept secret because they can be used to access protected information. Public keys are often widely distributed and often published on the open Internet. The intended use of a public key is that one can encrypt data with a downloaded public key corresponding to an intended recipient, encrypt sensitive information for the recipient with the public key, and send that encrypted sensitive information to the recipient. The recipient can use her secret key to decrypt and recover the protected information encoded in the plaintext.

Cryptographic algorithms are designed around computational hardness assumptions so that the difficulty of recovering information about plaintext is at least as hard as some provable computationally hard problem. Thus, it is theoretically possible to break such a system, but it is assumed infeasible to do so by any known practical means. Different computationally hard problems define different classes of encryption systems. PALISADE focuses on lattice-based cryptography. The security of lattice cryptography is based on the hardness of variants of the Shortest Vector Problem (SVP), Learning With Errors (LWE), and other hard problems. Lattice cryptography has both symmetric and public key variants, but we generally focus on the public key lattice cryptography variants in the PALISADE library and this manual.

Homomorphic Encryption (HE) or Fully Homomorphic Encryption (FHE) refer to a class of encryption methods envisioned by Rivest, Adleman, and Dertouzos in 1978. HE differs from basic encryption methods in that it allows computation to be performed directly on encrypted data without requiring access to a secret key. The result of such a computation remains in an encrypted form, and can at a later point be revealed by the owner of the secret key by decrypting the result.

In 2009 Craig Gentry showed the existence of lattice-based FHE capabilities. Since this initial discovery of a lattice-based FHE protocol, there has been a Renaissance in lattice cryptography, with the discovery of several other increasingly practical HE schemes and variations of HE protocols. Some of the common HE schemes include the Brakerski-Fan-Vercauteren (`BFV`)[1], Brakerski-Gentry-Vaikuntanathan (`BGV`), Cheon-Kim-Kim-Song (`CKKS`), `FHEW`, `TFHE` and Stehle-Steinfeld (`StSt`) schemes. Other related protocols include:

- Proxy Re-Encryption (`PRE`), which permits delegation of ciphertext decryption, thus allowing a host to delegate access to encrypted data.
- Somewhat Homomomorphic Encryption (`SHE`), which permits a limited amount of computation on encrypted data.
- Leveled Somewhat Homomorphic Encryption (Leveled SHE), which permits at least a fixed depth of computation to be performed on encrypted data by using a decreasing ladder (set) of ciphertext moduli.
- Multiparty Homomorphic Encryption, which enables multiple participants to contribute data to a joint computation, without sharing access to the actual data.

The public version of PALISADE supports all of the protocols discussed above, and we are in the process of adding support for more protocols and schemes. From v1.4 and onwards, PALISADE also supports a selection of trapdoor-based schemes: a Gentry-Peikert-Vaikuntanathan (`GPV`) digital signature scheme, a GPV identity-based encryption (`IBE`) scheme, and a Zhang-Zhang ciphertext-policy attribute-based encryption (`CP-ABE`) scheme.

An inherent property of encrypted computing technologies, including the various protocols supported by PALISADE, is that computing on encrypted data is significantly slower and more compute-intensive than computing on plaintext data. As such, debugging the applications of encrypted computing technologies can be a frustrating, slow process. To aid developers in integrating PALISADE implementations of encrypted computing technoligies, we also provide in PALISADE a "`Null`" scheme which supports the same API as the `BFV`, `BGV`, and `StSt` implementations, but which does not encrypt data and performs all operations on unencrypted plaintext. The `Null` scheme implementation operates as a light-duty no-security equivalent of the encrypted computing protocols supported by PALISADE. We provide the Null scheme so developers can test the correctness their PALISADE program more easily without the overhead or frustration of testing operations with slower more compute-intensive workloads engendered by encrypted computing capabilities.

[1] This scheme is also frequently denoted as the `FV` scheme, but we choose to call it the `BFV` scheme.

4 Library Architecture

The PALISADE library implements lattice cryptography in C++. The objects that are created by and manipulated within PALISADE are instances of C++ classes.

PALISADE is designed as a layered architecture where each layer provides a set of services to the layer "above" it in the stack, and makes use of services in the layer "below" it in the stack. The interfaces between each of the layers are designed to implement a common API. This permits substituting multiple implementations at any layer for experimental purposes.

The high-level architecture of PALISADE is illustrated in Figure 1.

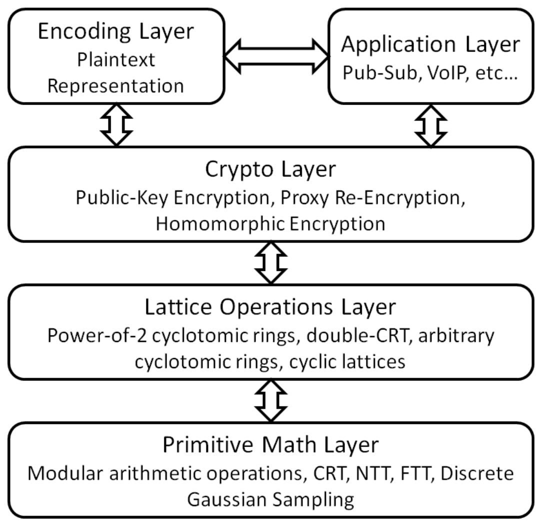

Figure 1: High-level PALISADE architecture

The layers in PALISADE are as follows:

1. **Application**: All programs that call the PALISADE library services are in this layer.
2. **Encoding**: All implementations of methods to encode data are at this layer.
3. **Crypto**: All implementation of cryptographic protocols are at this layer.
4. **Lattice Operations**: All higher-level lattice-crypto mathematical building blocks are in this layer.
5. **Primitive Math**: All low-level generic mathematical operations, such as multi-precision arithmetic implementations, are at this layer.

4.1 Application Layer

All programs that make use of the PALISADE library are said to be in the Application layer.

All programs at the Application layer make calls to services exposed in the PALISADE Crypto layer to gain access to PALISADE lattice cryptography functionality. The Application layer also makes use of service at the Encoding layer.

4.2 Encoding Layer

The Encoding layer contains all classes needed to provide for the encoding of any supported raw plaintext message data into a `Plaintext` object, and for decoding back to raw plaintext messages whenever necessary.

The Encoding layer is used to create `Plaintext` objects. These objects are sometimes created at the request of the Application layer, in which case Crypto layer methods act as a proxy for the Application layer, sometimes at the request of the Crypto layer itself. The interface for Encoding functionality is C++ methods provided by `Plaintext`.

4.3 Crypto Layer

The Crypto layer contains all classes needed to provide all available lattice cryptography functionality for specific cryptographic protocols such as public-key encryption, PRE and SHE schemes, and all included methods such as encryption and decryption, proxy re-encryption and container classes for parameters specific to those schemes.

The interface for Crypto functionality is C++ methods provided by `CryptoContext`. Therefore the Crypto layer provides various factory methods for creating a `CryptoContext` along with the context itself. The Crypto layer manipulates `Plaintext` and `Ciphertext` objects that are passed to it by the Application layer, and returns the appropriate `Plaintext` and `Ciphertext` objects to the Application. Operations on a `Matrix` of such objects is also provided.

Similar to `CryptoContext`, the library also provides `BinFHEContext`, `SignatureContext` and `ABEContext` for Boolean-circuit FHE, digital signature and GPV IBE/CP-ABE operations, respectively. However, unlike `CryptoContext` these contextes have their own implementation of ciphertexts. `SignatureContext` and `BinFHEContext` also use their own implementation of plaintexts. The Crypto layer makes use of services provided by the Lattice Operations layer.

4.4 Lattice Operations Layer

The Lattice Operations layer provides support for all lattice constructs, including power-of-two cyclotomic rings and arbitrary cyclotomic rings. The Double Chinese Remainder Theorem (Double-CRT) representation of cyclotomic rings is also implemented in the Lattice layer.

With the release of version 1.4.0, the lattice layer also hosts trapdoors and trapdoor sampling. Trapdoors are lattice constructs consisting of two vectors of polynomials. These structures are key parts of the newly added digital signature, identity-based encryption and ciphertext-policy attribute-based encryption schemes. These algorithms require a preimage of the information to be sampled from a discrete Gaussian over a lattice, which is done using the trapdoor sampling algorithms.

The Lattice layer is used to provide an implementation of the various `Poly` classes, including `Poly`, `NativePoly`, and `DCRTPoly`. These objects are used as building blocks in `Plaintext` and `Ciphertext`. The interface for Lattice functionality is the C++ methods provided by the `Poly` classes.

The Lattice layer performs lattice operations by decomposing operations into primitive arithmetic operations on integers, vectors, and matrices. The Lattice layer makes use of the Primitive Math layer to perform these operations.

4.5 Primitive Math Layer

The Primitive Math layer provides support for basic modular arithmetic operations, including multi-precision arithmetic. This layer also includes efficient algorithms for Number-Theoretic Transform (NTT), Fermat-Theoretic Transform (FTT), and discrete Gaussian samplers, among others. The interface for Math functionality is the C++ methods provided by both custom multi-precision libraries, and external imported libraries such as NTL.

4.6 Utilities

PALISADE also provides a cross-cutting Utilities module for common utility functions used by all the layers. The primary contents of this layer are tools for basic hashing, serialization and deserialization of objects, efficient memory allocation, debugging, and exception handling.

5 Capabilities

One of the goals of PALISADE is to provide straightforward mechanisms to select 1) an encryption protocol, 2) an encryption scheme to support that protocol, 3) encoding mechanisms to represent data with that scheme, 4) math back-ends to support efficient computational operations over that data for the scheme and 5) configuration parameters for that protocol, encoding mechanisms and math back-end. PALISADE provides a broad set of selections for each of these choices, and provides the user with the ability to add their own schemes, encodings, operations, etc. Furthermore, the user of PALISADE can select from multiple lattice layer implementations and math layer implementations, can easily mix-and-match, and can even provide their own implementations at different layers. In this section we describe these choices we make available in PALISADE.

At the highest level, the PALISADE library groups encryption protocols into a set of capabilities that can be selectively enabled by the user at run time. Enabling a particular capability turns on several functions in support that capability. Some capabilities imply other capabilities, such as how `SHE`, for example, implies `PKE` capabilities. Thus, in these scenarios, some capabilities are turned "on" automatically.

Capabilities and functions currently supported by PALISADE are listed in Tables 1–5. More detailed information on each function can be found in the PALISADE API specifications published in the gitlab repository (the API specifications are generated using Doxygen).

Table 1: Non-FHE PALISADE Capabilities and Functions

Capability	**Description**	**Functions Supported**	**Also Enabled**
ENCRYPTION	Public/Symmetric Key Encryption	KeyGen Encrypt Decrypt	
PRE	Proxy Re-Encryption	ReKeyGen ReEncrypt	PKE

Once an encryption protocol is selected, a user of PALISADE can select from one of multiple schemes:

- `BFV` variants
  - `BFV`: "textbook" Fan-Vercauteren variant of Brakerski's scale invariant scheme [Bra12, FV12, LN14]
  - `BFVrns`: Halevi-Polyakov-Shoup full RNS variant of the BFV scheme [HPS18, BPA+18]
  - `BFVrnsB`: Bajard-Eynard-Hasan-Zucca full RNS variant of the BFV scheme [BEHZ16, BPA+18]
- `BGV` variants
  - `BGV`: Brakerski-Gentry-Vaikuntanathan scheme [BGV14, GHS12]
  - `BGVrns`: A full RNS variant of the BGV scheme [KPZ20]
- `CKKS`: 2 full RNS variants of the Cheon-Kim-Kim-Song scheme [CKKS16] with various RNS optimizations from [CHK+18, BGP+19, HK19, KPP20]
- `StSt`: Stehle-Steinfeld scheme [SS11]
- `FHEW`: FHEW and extended TFHE schemes [DM14, CGGI18, MP20]
- `Null`: A system with all encryptions removed and all math operations done on `Plaintext` for development. Emulates BFV* and BGV* schemes (integer arithmetic operations).

Table 6 maps which schemes support which capabilities.

Starting with version 1.7, PALISADE also supports FHEW [DM14, MP20], a scheme for evaluating arbitrary Boolean circuits that runs bootstrapping for each binary gate evaluation. This scheme has very specialized functionality for working with boolean operations. Details on FHEW capabilities are provided in Table 5. Starting with release 1.8, PALISADE also supports FHEW with Gama-Izabachene-Nguyen-Xie (GINX) bootstrapping [GINX16], which is considered as an extended version of the TFHE scheme [MP20].

In addition to the homomorphic encryption schemes, PALISADE supports the following trapdoor-based schemes

- `GPV Signature`: Gentry-Peikert-Vaikuntanathan Digital Signature Scheme [GPV08, GPR+17]
- `GPV IBE`: Gentry-Peikert-Vaikuntanathan Identity-based Encryption Scheme [GPV08, GPR+17]
- `CP-ABE`: Zhang-Zhang Ciphertext-policy Attribute-based Encryption Scheme [ZZG12, GPR+17]

The functions for these schemes are listed in Table 7.

Table 2: PALISADE Operations/Functions included in the SHE capability (the Stehle-Steinfeld scheme is not included here as it supports only very limited/shallow computations). Note: BFV* and BGV* indicate all variants of BFV and BGV, respectively.

Functions Supported	Schemes Supported
EvalAdd	BGV*, BFV*, CKKS
EvalAdd (Scalar)	CKKS
EvalAdd (Plaintext)	BGV*, BFV*, CKKS
EvalAddMutable(All)	CKKS, BGVrns
EvalMult	BGV*, BFV*, CKKS
EvalMult (Scalar)	CKKS, BGVrns
EvalMult (Plaintext)	BGV*, BFV*, CKKS
EvalMultMutable (All)	CKKS, BGVrns
EvalMultAndRelinearize	BFV*, BGVrns, CKKS
EvalSub	BGV*, BFV*, CKKS
EvalSub (Scalar)	CKKS
EvalSub (Plaintext)	BGV*, BFV*, CKKS
EvalSubMutable (All)	CKKS, BGVrns
EvalNegate	BGV*, BFV*, CKKS
EvalLinearWSum(Mutable)	CKKS, BGVrns
EvalAutomorphism	BGV*, BFV*, CKKS
EvalAtIndex	BGV*, BFV*, CKKS
EvalFastRotation	CKKS, BGVrns
EvalSumRows	CKKS
EvalSumCols	CKKS
EvalSum	BGV*, BFV*, CKKS
EvalMerge	BGV*, BFV*, CKKS
EvalAddMany(InPlace)	BGV*, BFV*, CKKS
EvalMultMany	BGV*, BFV*, CKKS
EvalInnerProduct	BGV*, BFV*, CKKS
EvalLinRegression	BGV*, BFV*, CKKS
EvalLinRegressBatched	BGV*, BFV*, CKKS
EvalCrossCorrelation	BGV*, BFV*, CKKS
EvalRightShift	BGV*, BFV*, StSt
———	———
KeySwitch	BGV*, BFV*, CKKS
Relinearize	CKKS, BGVrns
KeySwitchGen	BGV*, BFV*, CKKS
EvalMultKeyGen	BGV*, BFV*, CKKS
EvalAutomorphismKeyGen	BGV*, BFV*, CKKS
EvalAtIndexKeyGen	BGV*, BFV*, CKKSl
EvalSumKeyGen	BGV*, BFV*, CKKS
EvalSumRowsKeyGen	CKKS
EvalSumColsKeyGen	CKKS
AdjustLevelWithRescale	CKKS
EvalFastRotationPrecompute	CKKS, BGVrns
AddRandomNoise	BGV*, BFV*, CKKS

Table 3: PALISADE Operations/Functions included in the `LeveledSHE` capability. Note: BGV* indicates all variants of BGV

Functions Supported	Schemes Supported
ModReduce	BGV*
Rescale	CKKS
LevelReduce	CKKS
Compress	BGVrns, CKKS
EvalPoly	CKKS

Table 4: PALISADE Operations/Functions included in the `MULTIPARTY` (Threshold FHE) capability. Note: BFV* and BGV* indicate all variants of BFV and BGV, respectively

Functions Supported	Schemes Supported
MultiKeySwitchGen	BGV*, BFV*, CKKS
MultiEvalAutomorphismKeyGen	BGV*, BFV*, CKKS
MultiEvalSumKeyGen	BGV*, BFV*, CKKS
MultiEvalAtIndexKeyGen	BGV*, BFV*, CKKS
MultiAddPubKeys	BGV*, BFV*, CKKS
MultiAddEvalKeys	BGV*, BFV*, CKKS
MultiMultEvalKey	BGV*, BFV*, CKKS
MultAddEvalSumKeys	BGV*, BFV*, CKKS
MultiAddEvalAutomorphismKeys	BGV*, BFV*, CKKS
MultiAddEvalMultKeys	—
—	BGV*, BFV*, CKKS
MultipartyDecryptLead	BGV*, BFV*, CKKS
MultipartyDecryptMain	BGV*, BFV*, CKKS
MultipartyDecryptionFusion	BGV*, BFV*, CKKS

Table 5: PALISADE Capabilities and Functions for Boolean-Circuit FHE (performs bootstrapping for each binary gate evaluation)

Capability	Description	Functions Supported
`FHEW`	FHE Scheme for Arbitrary Boolean Circuits	KeyGen BTKeyKeyGen Encrypt Decrypt Bootstrap EvalNOT EvalBinGate(OR) EvalBinGate(AND) EvalBinGate(NOR) EvalBinGate(NAND) EvalBinGate(XOR) EvalBinGate(NXOR) EvalBinGate(XOR_FAST) EvalBinGate(NXOR_FAST)

Table 6: PALISADE Implemented Capabilities Matrix

Function/Scheme	**CKKS**	**StSt**	**BGV***	**BFV***	**Null**
ENCRYPTION	Y	Y	Y	Y	Y
PRE	Y	Up to 2 hops	Y	Y	Y
SHE	Y	Up to 2 levels	Y	Y	Y
LeveledSHE	Y		Y		Y
Multiparty	Y	Limited	Y	Y	Y

Table 7: Other PALISADE Capabilities and Functions

Capability	**Description**	**Functions Supported**	**Also Enabled**
CP-ABE	Ciphertext-Policy Attribute-Based Encryption	Setup KeyGen KeyGenOfflinePhase KeyGenOnlinePhase Encrypt Decrypt	
GPV Signature	Digital Signature	KeyGen Sign SignOfflinePhase SignOnlinePhase Verify	
GPV IBE	Identity-Based Encryption	Setup KeyGen KeyGenOfflinePhase KeyGenOnlinePhase Encrypt Decrypt	

After the selection of the scheme, a user can then select how to represent polynomials, among the following polynomial types:

- `Poly`, which corresponds to representing polynomial coefficients in a multiprecision integer format.
- `DCRTPoly`, which corresponds to representing polynomial coefficients in a Residue Number System (RNS), a.k.a. Chinese Remainder Theorem (CRT), format using single-precision integers.
- `NativePoly`, which corresponds to representing polynomial coefficents in a single-precision format (only works up to the word size of 60 bits).

We generally recommend using the `DCRTPoly` representation of polynomials for performance on commodity computing environments. We include the multiprecision polynomial representation `Poly` primarily for advanced users and implementers of new schemes who wish to do an initial implementation using the simpler, less efficient `Poly` representation.

Similar to the polynomial representations, PALISADE also supports multiple big integer math back-ends. The user selects which of the various math back-ends are included at build or compile time [2]. Currently supported multi-precision math back-ends include:

- Multi-precision with fixed bitwidth sizing (`MATHBACKEND 2`) supporting bit widths up to 3500 by default (this limit can be changed in a header file). This is the default back-end.
- Multi-precision with dynamic bitwidth sizing (`MATHBACKEND 4`). This back-end is slightly slower than `MATHBACKEND 2`, but allows for extremely large integers.
- `NTL` and `GMP` based multi-precision with variable sizing and optimization for various platforms (`MATHBACKEND 6`) [3]

Note the numbering of back-ends is due to historical reasons, as various back-ends have been deprecated over time.

Native data types are always available (and used for `NativePoly` and single-precision integers in `DCRTPoly`), and we currently support native 64-bit and 128-bit word sizes. This limitation may be lifted in future releases, allowing PALISADE to be used on 32-bit architectures.

[2] Math back-end selection is controlled at build setup time by adding a `cmake` flag, e.g., `-DMATHBACKEND=4`, or at compile time by editing `src/core/lib/math/backend.h`.

[3] We include the minimum required version of the `GMP` and `NTL` distribution with our release.

6 PALISADE Directory Structure

The directory structure of the PALISADE source code is shown in Listing 1 and Table 8.

```
|- benchmark
|- build
|- doc
|- src
|  |- binfhe
|  |  |- include    -- Boolean-circuit FHE header files
|  |  |- lib    -- Boolean-circuit FHE source files
|  |- core
|  |  |- include    -- PALISADE core header files
|  |     |- math
|  |     |- lattice
|  |     |- encoding
|  |     |- util
|  |  |- lib    -- PALISADE core source files
|  |     |- math
|  |     |- lattice
|  |     |- encoding
|  |     |- util
|  |- abe
|  |  |- include    -- Attribute-based encryption crypto layer header files
|  |  |- lib    -- Attribute-based encryption crypto layer source files
|  |- pke
|  |  |- include    -- Homomorphic scheme header files
|  |     |- scheme
|  |        |- bfv -- Original BFV (integ. arithm. - slower than RNS variants)
|  |        |- bfvrns -- HPS RNS variant of BFV scheme (integer arithmetic)
|  |        |- bfvrnsb -- BEHZ RNS variant of BFV scheme (integer arithmetic)
|  |        |- bgv -- BGV scheme (integer arithmetic)
|  |        |- bgvrns -- full RNS variant of BGV scheme (integer arithmetic)
|  |        |- ckks -- CKKS scheme (real-number arithmetic)
|  |        |- null -- NULL scheme (integer arithemtic)
|  |        |- stst -- Stehle-Steinfeld scheme (limited  HE operations)
|  |  |- lib    -- Public key encryption crypto layer source files
|  |     |- scheme
|  |        |- bfv -- Original BFV (integ. arithm. - slower than RNS variants)
|  |        |- bfvrns -- HPS RNS variant of BFV scheme (integer arithmetic)
|  |        |- bfvrnsb -- BEHZ RNS variant of BFV scheme (integer arithmetic)
|  |        |- bgv -- BGV scheme (integer arithmetic)
|  |        |- bgvrns -- full RNS variant of BGV scheme (integer arithmetic)
|  |        |- ckks -- CKKS scheme (real-number arithmetic)
|  |        |- null -- NULL scheme (integer arithemtic)
|  |        |- stst -- Stehle-Steinfeld scheme (limited  HE operations)
|  |- signature
|     |- include    -- Signature crypto layer header files
|     |- lib    -- Signature crypto layer source files
|- test         -- Test software scripts
|- third-party  -- External third-party software
```

Listing 1: Directory structure for PALISADE source code

All of the src/core, src/pke. src/binfhe, src/abe and src/signature contain subdirectories for header files (`include`), library source files (`lib`), unit tests (`unittest`) as well as

Table 8: PALISADE Library File Structure

Directory	Description
benchmark	Code for benchmarking PALISADE library components.
build	Binaries and build scripts (this folder is created by the user).
doc	Documentation of library components, including doxygen.
src	Library source code.
test	Google unit test code.
third-party	Code provided by an external third party.

subdirectories for main examples (`examples`) that exercise most common capabilities of PALISADE and additional examples (`extras`) for less common scenarios.

7 Terminology and Notation

In this section we provide a glossary of terminology that we use in PALISADE.

PALISADE provides a framework for using lattice cryptography.

If a `Scheme` supports the ENCRYPTION `Capability`, encrypt and decrypt methods are available for use. The encrypt and decrypt methods are provided as part of the implementation of the ENCRYPTION `Capability` in the `Scheme` being used. A `Plaintext` can be encrypted into a `Ciphertext` through the use of an encrypt method. A `Ciphertext` can be used to generate a `Plaintext` through the use of a decrypt method. The library supports several formats of `Plaintext` and methods to create a `Plaintext` for use in PALISADE.

If a `Scheme` supports the PRE `Capability`, a re-encrypt method is available.

If a `Scheme` supports the SHE and/or LeveledSHE `Capability`, then there is support for homomorphic operations on pairs of `Ciphertext` as well as mixed-mode operations between a `Ciphertext` and a `Plaintext`. These operations are supported by using C++ operator overloading. For example, multiplication of two instances of a `Ciphertext`, `A` and `B`, can be simply written as `A * B`.

PALISADE also includes support for several `Matrix` operations. It is possible to create several `Matrix` of `Plaintext`, encrypt them, perform operations on them, and then decrypt back into a `Matrix` of `Plaintext`.

7.1 Typing

All PALISADE operations are strongly typed. A `Plaintext` that is passed to encrypt will create a `Ciphertext` that is aware of the underlying format of the `Plaintext`, as well as the particular key that was used to encrypt the `Plaintext`. The decrypt operation will fail in cases where an improper key is used. A successful decrypt will produce a new `Plaintext` whose underlying format matches the initial `Plaintext` that was passed to encrypt. Homomorphic operations between `Ciphertexts`, and mixed-mode operations between a `Ciphertext` and a `Plaintext` will only be permitted for operands with formats and keys that match.

The underlying data inside of PALISADE is an `Element` in lattice space. Within PALISADE, all `Plaintexts` are encoded into an `Element`. Every `Ciphertext` contains one or more `Element` objects, and all operations are mathematical operations on these `Elements`.

There are several formats of `Element` available: a `Poly`, which represents the encoded `Plaintext` as a polynomial; a `NativePoly`, which uses a polynomial with coefficients with a maximum size of 64 bits (when the native backend is set to use 64-bit integers); and a `DCRTPoly`, which represents the encoded `Plaintext` as a stack of decomposed `NativePoly` polynomials using the Double Chinese Remainder Theorem.

7.2 CryptoContext

A `CryptoContext` in PALISADE is the class that provides all PALISADE encryption functionality. All objects used in a PALISADE implementation are created by a `CryptoContext` and can be considered to "belong to" the `CryptoContext` that they were created with.

Any and all operations on PALISADE objects must be on objects that belong to the same `CryptoContext`. The high-level use of the `CryptoContext` to encrypt a `Plaintext` to generate a `Ciphertext` is as follows:

1. Choose a set of `ElementParams` to define parameters for the `Element` to be used.
2. Choose a set of `EncodingParams` to define parameters for encoding.
3. Select a `Scheme` that you wish to use for lattice cryptography.
4. Construct a `CryptoContext` for your selected scheme, `ElementParams` and `EncodingParams`, in which all operations shall take place. The construction of a `CryptoContext` involves selecting parameters for security and performance. There are potentially multiple mechanisms to generate the Scheme parameters needed for this construction.
5. Select which `Capability` are used with the `CryptoContext`. Note that not every `Scheme` will support every possible `Capability`.
6. Use CryptoContext methods to create `Keys`.
7. The user may also perform homomorphic operations on `Ciphertext` and `Plaintext` objects if that `Capability` has been enabled and if the `Scheme` supports the operations.
8. Encrypt a `Plaintext` into a `Ciphertext`.
9. Decrypt a `Ciphertext` back to a `Plaintext`.

Each `CryptoContext` is uniquely identified by its `Scheme`, `Element` type, `ElementParams` and `EncodingParams`. When we say that an object "belongs to" a `CryptoContext`, we are actually saying that it is associated with a `CryptoContext` with a particular `Scheme`, `Element` type, `ElementParams` and `EncodingParams`.

PALISADE incorporates the standard security tables developed during the Homomorphic Encryption standardization process described at `http://homomorphicencryption.org`. Users of the library can construct a `CryptoContext` by specifying the parameter sets defined in the standard.

It follows that if one creates a `CryptoContext` on two different computers, each with the same `Scheme`, `Element` type, `ElementParams` and `EncodingParams`, then those two `CryptoContexts` are the same, and objects created on one machine can be transferred to and used on the other machine.

7.3 BinFHEContext

A `BinFHEContext` is the equivalent of a `CryptoContext` for Boolean-circuit FHE. All operations are performed usng an instance of the `BinFHEContext` class. The high-level use of an `BinFHEContext` to perform FHE operations is as follows:

1. Choose a predefined configuration to instantiate the context. The options are:
   - `TOY` (Toy parameter set for unit tests and debugging),
   - `MEDIUM` ($\geq$ 100 bits of security for both classical and quantum computer attacks),
   - `STD128` (HE standard set with more than 128 bits of security w.r.t. classical computer attacks),
   - `STD192` (HE standard set with more than 192 bits of security w.r.t. classical computer attacks),
   - `STD256` (HE standard set with more than 256 bits of security w.r.t. classical computer attacks),
   - `STD128Q` (HE standard set with more than 128 bits of security w.r.t. quantum computer attacks),
   - `STD192Q` (HE standard set with more than 192 bits of security w.r.t. quantum computer attacks),
   - `STD256Q` (HE standard set with more than 256 bits of security w.r.t. quantum computer attacks).
2. Generate a secret key (an instance of `LWEPrivateKey`).
3. Generate the bootstrapping keys, including a refreshing and switching keys. The bootstrapping keys are stored inside the context.
4. Encrypt a plaintext (`LWEPlaintext`) to form a ciphertext (an instance of `LWECiphertext`).
5. Perform any of the following operations as many times as needed: `NOT` (no boostrapping), `OR`, `AND`, `NAND`, `NOR`, `XOR`, or `XNOR`. The last six require bootstrapping and take the same time to execute.
6. `Decrypt` the result using the secret key.

7.4 ABEContext

An `ABEContext` is the equivalent of a `CryptoContext` for attribute-based and identity-based encryption functionality. All `CP-ABE` and GPV `IBE` related operations are done using an `ABEContext`. The high-level use of an `ABEContext` to encrypt a `Plaintext` is as follows:

1. Choose a set of parameters to create appropriate set of `ElementParams` for the `Element` to be used.

2. Select a `Scheme` to use (GPV `IBE` or `CP-ABE`).

3. Construct a `ABEContext` for your selected scheme and parameters. Note that the steps up to this point are similar to `CryptoContext`, with the exception of `EncodingParams`. This is due to the fact that `ABEContext` at its current implementation only supports the plaintext modulus of 2 for encoding, hence it is done internally.

4. Create an `AccessPolicy` for the parties that will be able to decrypt.

5. Use `ABEContext` methods to create `Keys`.

6. Encrypt a `Plaintext` under an `AccessPolicy` into a `Ciphertext`.

7. `Decrypt` a `Ciphertext` back to a `Plaintext`.

   The standard security tables developed during the Homomorphic Encryption standardization process described at `http://homomorphicencryption.org` also apply to `ABEContext`.

   It is also worth to mention that `ABEContext` does not support `DCRTPoly` at its current state. However, for most cases `NativePoly` is enough. For the time being, `ABE` does not have homomorphic operations within ciphertexts defined. Because of this the ciphertext modulus required for the security/correctness of regular encryption functionality is below 64 bits.

7.5 SignatureContext

A `SignatureContext` is the equivalent of a `CryptoContext` for digital signature schemes. All GPV signature scheme-related operations are done using a `SignatureContext`. The high-level use of a `SignatureContext` to sign a `Plaintext` is as follows:

1. Choose a set of parameters to create appropriate set of `ElementParams` for the `Element` to be used.

2. Select a `Scheme` to use (GPV only for the time being).

3. Construct a `SignatureContext` for your selected scheme and parameters. Note that the steps up to this point are similar to `CryptoContext`, with the exception of `EncodingParams`. This is due to the fact that the encoding is handled internally.

4. Use SignatureContext methods to create `Keys`.

5. Sign a `Plaintext` into a `Signature`.

6. Verify a `Signature`with a `Plaintext`.

It is also worth to mention that just like `ABEContext`, `SignatureContext` does not support `DCRTPoly` at its current state and `NativePoly` is enough for all cases. Homomorphic operations are not defined for signatures, which means there is no need for moduli larger than 64 bits.

7.6 Plaintext

A `Plaintext` is used in PALISADE to represent something that has not been encrypted. It is actually the base class for each of the possible plaintext encodings that are supported in PALISADE:

- `ScalarEncoding`
- `IntegerEncoding`
- `FractionalEncoding`
- `PackedEncoding`
- `CKKSPackedEncoding`
- `CoefPackedEncoding`
- `StringEncoding`

A `Plaintext` is created by invoking the appropriate `CryptoContext` method, passing it the unencrypted information as a parameter.

Once created, a `Plaintext` can be encrypted into a `Ciphertext` using the `CryptoContext` Encrypt routine.

A `Plaintext` can also be used as a parameter to several of the `CryptoContext` homomorphic operations.

When a `Ciphertext` is decrypted, the decryption method creates a new `Plaintext` to contain the decryption.

The `Plaintext` has several methods that provide access to the information in the `Plaintext`, as shown in listing 2:

```
const std::string&              GetStringValue();
const int64_t                   GetIntegerValue();
const int64_t                   GetScalarValue();
const vector<int64_t>&          GetCoefPackedValue();
const vector<uint64_t>&         GetPackedValue();
const vector<complex<double>>&  GetCKKSPackedValue();
```

Listing 2: Values in Plaintext

Note that the only one of these `Plaintext` methods that will return a value is the one that corresponds to the actual type of the `Plaintext`. For example, GetStringValue() will return a std::string if the `Plaintext` is actually a `StringEncoding`, and will throw an exception otherwise.

While `ABEContext` also uses `Plaintext` for unencrypted information, `SignatureContext` uses its own plaintext definition.

7.7 Ciphertext

A `Ciphertext` is used in PALISADE to represent encrypted information. A `Ciphertext` is created from a `Plaintext` by an encrypt method, and is converted back to a `Plaintext` by a decrypt method. The encrypt and decrypt methods also require `Keys` that have been generated by the `CryptoContext`.

Homomorphic operations between pairs of `Ciphertext`, or between a `Ciphertext` and a `Plaintext`, are supported, provided that the encodings of the operands match, and provided that those encodings support homomorphic operations (for example, homomorphic operations are not supported for string encodings, and will be rejected if attempted).

The `CP-ABE`/GPV `IBE` schemes utilizing `ABEContext` have their own unique classes for ciphertext implementations, which do not have homomorphic operations supported at this time.

7.8 Access Policy

An `AccessPolicy` is a rule/set of rules dictating the decrypting party of the communication. User holding the appropriate rights can generate the relevant `Keys` for decryption and decrypt the information encrypted under an `AccessPolicy`.

In the context of GPV `IBE`, `AccessPolicy` refers to the unique user identifier.

In the context of `CP-ABE`, `AccessPolicy` can refer to both attributes defined for the access policy as well as user's own attribute set.

7.9 Signature

A `Signature` is used in PALISADE for authentication. A `Signature` is created from a `Plaintext` by a sign method, and used in conjunction with the `Plaintext` for verification. The sign and verify methods also require `Keys` that have been generated by the `SignatureContext`.

7.10 Keys

Many PALISADE functions make use of `Keys`. These objects are created using `CryptoContext`, `ABEContext` or `SignatureContext` methods.

Encryption and decryption functionality requires a public key/private key pair. The `CryptoContext` KeyGen method generates this key pair and returns it to the caller.

`CP-ABE` and GPV `IBE` operations require a master public key/private key pair for the encryption and derivation of personal secret keys. This is done using Setup method of `ABEContext`. For decryption, a secret key corresponding to an access policy/user is required, which is done using the `ABEContext KeyGen` method.

Signing and verification operations for digital signature schemes require a sign(private)/verify(public) key pair. This pair can be generated using the `SignatureContext KeyGen` method.

Re-Encryption requires an Evaluation key. This key is generated using the `ReKeyGen` method.

Certain homomorphic operations may require the application of evaluation keys to complete the operation. The `CryptoContext EvalMultKeyGen` and `EvalAddKeyGen` methods are used to generate the requisite keys needed for the `EvalMult` and `EvalAdd` operations, respectively.

7.11 Capability

A `Capability` refers to sets of `CryptoContext` methods that must be enabled before they can be used. A `Capability` must be enabled by calling the `CryptoContext` Enable method and passing it the name of the `Capability` as an argument. Multiple values for `Capability` can be or-ed together and passed as a single argument to Enable.

Section 5 discusses in the detail the available choices for `Capability`, and the functionality enabled when a particular `Capability` is Enabled.

7.12 Scheme

A `Scheme` (the full class name is `LPPublicKeyEncryptionScheme`) contains the algorithms used for key generation, encryption/decryption, and homomorphic operations.

Table 6 outlines the different supported schemes in PALISADE.

In order to use a particular `Scheme`, its configuration parameters must be specified. The particular configuration parameters for each scheme are stored in a `CryptoParameters` class (the full class name is `LPCryptoParameters`) that is particular for that scheme. The classes for `CryptoParameters` for each scheme are unique, but they all share several characteristics:

1. Each `CryptoParameters` class has a constructor that allows for the specification of all configuration parameters.

2. The `CryptoParameters` may have a `ParamsGen` feature that generates a full set of configuration parameters from a small specification as to how many homomorphic operations will be performed.
3. The `CryptoParameters` class is part of the `CryptoContext`

`ABEContext` and `SignatureContext` also use their equivalent parameter classes like `CryptoParameters`.

7.13 Element

The math layer performs operations on different kinds of `Element`. This is a representation of a vector in lattice space.

7.13.1 Poly

A `Poly` is a vector of polynomial coefficients. The coefficients are whatever the `BigInteger` type is for the selected math back-end, and the vector is simply a vector of these `BigInteger`, and an associated `modulus`. All operations on a `Poly` are done modulo this `modulus`.

7.13.2 NativePoly

A `NativePoly` is a vector of polynomial coefficients where each of the coefficients is of type 64-bit unsigned integer. The `NativePoly` also has a `modulus` of at most 64 bits (though our math implementations are most efficient for moduli of 60 bits or less). Again,all operations are done modulo the `modulus`.

7.13.3 DCRTPoly

A `DCRTPoly` implements a large Poly decomposed via residue arithmetic into a tower (set) of `NativePoly` elements.

7.14 ElementParams

An `ElementParams` is a container for the configuration parameters of whatever Element is being used. The configuration parameters include the ring dimension, cyclotomic order, and primitive root of unity.

7.15 EncodingParams

An `EncodingParams` is a container for all of the parameters needed to encode a `Plaintext`. In most cases, this consists solely of a plaintext `modulus`; however, some encodings require more detailed parameters.

7.16 Matrix

PALISADE supports general operations `Matrix` objects with elements of the types described above in this section, as well as operations on these matrices. A user can create a matrix of `Plaintext`, encrypt it into a matrix of `Ciphertext`, and perform operations on matrices.

In addition to homomorphic matrix multiplication, higher level statistical operations such as inner product and linear regression are available. Some operations result in a matrix of `RationalCiphertext`, where each entry in the matrix is a rational number such that the numerator and denominator of the entry is a `Ciphertext`. For matrices of `RationalCiphertext`, separate decryption of the numerators and denominators of each entry are provided.

8 Sample Implementations: PKE Module

8.1 Creating a CryptoContext

All PALISADE operations in the PKE module are associated with a CryptoContext; therefore, the first step in using PALISADE is to acquire a CryptoContext. A CryptoContext can be created in a number of different ways through the use of static CryptoContextFactory methods. The factory methods return a shared_ptr to a CryptoContext.

It is useful to note that PALISADE keeps track all of the CryptoContexts that have been created, and will not create duplicate contexts. If a user requests that the factory create a context that already exists, the factory simply returns a shared_ptr to the existing context rather than creating a duplicate of that existing context.

A CryptoContext is uniquely identified by the parameters for the underlying lattice layer element that is to be used, the encoding parameters to be used with Plaintext, and the Scheme (and any associated configuration parameters for the Scheme).

The underlying lattice layer element is either a Poly, a NativePoly, or a DCRTPoly. These type names are used as template parameters with the CryptoContext and its methods. The lattice layer elements share a set of element parameters that are given to the CryptoContextFactory methods.

Encoding parameters may be a simple Plaintext modulus, or they may be a broader set of EncodingParms that are used. Encoding parameters are passed to the CryptoContextFactory methods.

Different Schemes have wildly different configuration parameters, and so the Scheme is selected by calling a CryptoContextFactory method for the desired Scheme.

Each Scheme actually has several factory methods:

- A method that accepts values for all configuration parameters for the scheme.
- A method that accepts some values for configuration parameters, and that invokes a scheme-specific parameter generation operation.

There are two other mechanisms available for creating a CryptoContext:

- Deserialize a previously serialized CryptoContext.
- Construct a CryptoContext from one of the set of predefined scheme parameters.

Once a CryptoContext has been created, the appropriate capabilities should be enabled. Once this is complete, the CryptoContext is ready for use.

Below are some code samples for creating contexts. Listing 3 demonstrates creating a CryptoContext by specifying all of the scheme's parameters.

```
/// Example showing creating a BGV CryptoContext
/// By specifying all Scheme parameters
/// Element is Poly with non power-of-two cyclotomics

usint m = 22;
PlaintextModulus p = 2333;
BigInteger ptm(p);

BigInteger ctm("955263939794561");
BigInteger srr("941018665059848");
BigInteger bigmod("80899135611688102162227204937217");
BigInteger bigroot("77936753846653065954043047918387");

auto cycloPoly = GetCyclotomicPolynomial<BigVector>(m, ctm);
ChineseRemainderTransformArb<BigInteger, BigVector>::SetCylotomicPolynomial(
    cycloPoly, ctm);

float stdDev = 3.19;
usint bSize = 8;

shared_ptr<Poly::Params> params(new ILParams(m, ctm, srr, bigmod, bigroot));

EncodingParams ep(new EncodingParamsImpl(p, PackedEncoding::
    GetAutomorphismGenerator(p), bSize));

CryptoContext<Poly> cc = CryptoContextFactory<Poly>::genCryptoContextLTV(
    params, ep, bSize, stdDev);

cc->Enable(ENCRYPTION);
cc->Enable(SHE);
```

Listing 3: Creating a CryptoContext with parameters

Listing 4 demonstrates the use of the factory method that uses parameter generation.

Listing 5 shows the use of preconfigured parameter sets.

```
/// Example showing creating a BFV Cryptocontext
/// using parameter generation
/// Element is DCRTPoly (configured by ParamsGen)

int relWindow = 1;
PlaintextModulus ptm = 256;
double sigma = 4;
double rootHermiteFactor = 1.006;

//Set Crypto Parameters
CryptoContext<Poly> cryptoContext =
        CryptoContextFactory<Poly>::genCryptoContextBFV(
                ptr, rootHermiteFactor, relWindow, sigma, 0, 5, 0, OPTIMIZED,
                    6);

// enable features that you wish to use
cryptoContext->Enable(ENCRYPTION);
cryptoContext->Enable(SHE);
```

Listing 4: Creating a CryptoContext with parameter generation

```
/// Example showing CryptoContext generation
// from a preconfigured parameter set

CryptoContext<Poly> cryptoContext =
        CryptoContextHelper::getNewContext("BGV3");

if( !cryptoContext ) {
        cout << "Error creating CryptoContext" << endl;
        return 0;
}

cryptoContext->Enable(ENCRYPTION);
```

Listing 5: Creating a preconfigured CryptoContext

Listing 6 shows the use of parameter sets defined by the Homomorphic Encryption standardization process, as defined in `http://homomorphicencryption.org`.

```
/// Example showing CryptoContext generation
// using standard tables from the homomorphic encryption standardization
   project


PlaintextModulus ptm = 536903681;
double sigma = 3.2;
SecurityLevel securityLevel = HEStd_128_classic;

// support 7 multiplies (ciel(log2(7)))
usint nMults = 3;

// generate keys for s^2 and s^3
usint maxDepth = 3;

CryptoContext<DCRTPoly> cryptoContext =
     CryptoContextFactory<DCRTPoly>::genCryptoContextBFVrns(
          ptm, securityLevel, sigma, 0, nMults, 0, OPTIMIZED, maxDepth);

if( !cryptoContext ) {
        cout << "Error creating CryptoContext" << endl;
        return 0;
}

cryptoContext->Enable(ENCRYPTION);
```

Listing 6: Creating a CryptoContext using standard paranerers

Listing 7 shows creation from a serialization.

```
/// Example showing CryptoContext generation
/// from a serialization

// Deserialize the crypto context
CryptoContext<DCRTPoly> cc;
if ( !Serial::DeserializeFromFile(DATAFOLDER + "/cryptocontext.txt", cc,
   SerType::BINARY) ) {
        cerr << "I cannot read serialization from " << DATAFOLDER + "/
           cryptocontext.txt" << endl;
}
else
        cout << "The cryptocontext has been deserialized." << std::endl;
```

Listing 7: Creating a CryptoContext from a serialization

8.2 Creating A Plaintext

PALISADE users are able to convert integers, vectors of integers, vectors of complex numbers, and strings into Plaintext objects.

Plaintext objects can be used as input to Encryption, can be used as part of homomorphic operations, and are produced as a result of decryption.

All Plaintexts are created by static factory methods within the CryptoContext. Each style of Plaintext encoding has its own factory method.

The Plaintexts are all type safe. A Ciphertext knows the encoding used by the original Plaintext that it came from. Decryption creates a Plaintext with the proper encoding. All homomorphic operations check types before performing the operation, and will fail if either the encodings do not match, or if a particular encoding does not support homomorphic operations.

8.2.0.1 ScalarEncoding

ScalarEncoding is used to encode a single integer by simply copying the integer into the polynomial starting at index 0 and zero-filling remaining unused indices of the polynomial. Listing 8 shows an example of creating a Scalar Plaintext.

```
int64_t val = 12;

Plaintext sPtx = ctx->MakeScalarPlaintext(val);

Plaintext sPtx2 = ctx->MakeScalarPlaintext(-val);
```

Listing 8: Creating a Scalar Plaintext

8.2.0.2 IntegerEncoding

IntegerEncoding is used to encode a single unsigned integer into a polynomial such that each bit (0 or 1) in the integer is copied into its corresponding slot in the polynomial. For example, the integer 14, which is binary 1110, is encoded by emplacing 1, 1, 1 and 0 into the first four positions of the polynomial. This is shown in listing 9.

```
Plaintext intPtx = ctx->MakeIntegerPlaintext(14);
```

Listing 9: Creating an Integer Plaintext

Important Note: IntegerEncoding encodes each bit of the integer into a separate coefficient of the polynomial. This implies that the ring dimension (degree of polynomial) should be large enough to store all bits, especially in scenarios where each homomorphic multiplication doubles the number of polynomial coefficients with each multiplication. This is typically a non-issue for secure schemes/settings (as the ring dimension is already large enough) but can be an issue for the Null scheme, where the cyclotomic order is provided as an input argument.

8.2.0.3 FractionalEncoding

FractionalEncoding is a generalization of IntegerEncoding that can encode both integers and fractions. Currently it is limited in its functionaly and is only used to add support for right shifting (moving least significant bits to the fractional part). Examples of FractionalEncoding for encoding an integer 3 and a fraction 1/8 are shown in listing 10.

```
Plaintext intPtx = ctx->MakeFractionalPlaintext(3);
Plaintext intPtx = ctx->MakeFractionalPlaintext(0,3);
```

Listing 10: Creating a Fractional Plaintext

Important Note: FractionalEncoding encodes each bit of the integer into a separate coefficient of the polynomial. This implies that the ring dimension (degree of polynomial) should be large enough to store all bits, especially in scenarios where each homomorphic multiplication doubles the number of polynomial coefficients with each multiplication. This is typically a non-issue for secure schemes/settings (as the ring dimension is already large enough) but can be an issue for the Null scheme, where the cyclotomic order is provided as an input argument.

8.2.0.4 CoefPackedEncoding

CoefPackedEncoding is used to encode a vector of integers, or an initializer list of integers, into a polynomial such that each integer is emplaced into a coefficient of the polynomial. This is illustrated in listing 11.

```
std::vector<int64_t> inputs( { 2, 3, 5, 7} );

// construction from vector
Plaintext c1 = ctx->MakeCoefPackedPlaintext(inputs);

// construction from initializer list
Plaintext c2 = ctx->MakeCoefPackedPlaintext({-6,1,4,8});
```

Listing 11: Creating a CoefPackedEncoding Plaintext

8.2.0.5 PackedEncoding

PackedEncoding is an implementation of an efficient encoder packing multiple integers into a single plaintext polynomial (single ciphertext) that enables SIMD (Single Instruction, Mutiple Data) operations on these integers. Currently PALISADE supports addition, multiplication, and rotation capabilities for packed ciphertexts. The cyclotomic order m has to divide $p-1$, where p is the plaintext modulus. This is shown in listing 12.

```
std::vector<int64_t> val( {37, 22, 18, 4, 3, 2, 1, 9} );

Plaintext packedPtx = ctx->MakePackedPlaintext(val);
```

Listing 12: Creating a PackedEncoding Plaintext

8.2.0.6 CKKSPackedEncoding

CKKSPackedEncoding is an implementation of an efficient encoder packing multiple complex/real numbers into a single plaintext polynomial (single ciphertext) that enables SIMD (Single Instruction, Mutiple Data) operations on these complex/real numbers. Currently PALISADE supports addition, multiplication, and rotation capabilities for packed ciphertexts. An exampple is shown in listing 13.

```
// a vector of real numbers
std::vector<complex<double>> val( {0.1111, 0.2222, 0.3333, 0.4444, 0.5555,
    0.6666, 0.7777, 0.8888} );
Plaintext packedPtx = ctx->MakeCKKSPackedPlaintext(val);

// a vector of complex numbers
std::vector<complex<double>> valComplex( {0.1111 + 0.1i, 0.2222 + 0.2i, 0.3333
     + 0.1i, 0.4444 + 0.1i, 0.5555 + 0.1i, 0.6666 + 0.1i, 0.7777 + 0.1i,
    0.8888 + 0.1i} );
Plaintext packedPtxComplex = ctx->MakeCKKSPackedPlaintext(valComplex);
```

Listing 13: Creating a CKKSPackedEncoding Plaintext

8.2.0.7 StringEncoding

StringEncoding is used to encode a string into a polynomial. For this encoding, each 8-bit character in the input is encoded directly into a coefficient of the polynomial. This encoding is constrained to only use 256 as a plaintext modulus. Listing 14 shows the use of this encoding.

Future implementations that support other character encodings, such as Unicode, are possible but are not currently supported.

Homomorphic operations will NOT work with this encoding.

```
string s("Here is my string!");

Plaintext stringPtx = ctx->MakeStringPlaintext(s);
```

Listing 14: Creating a String Plaintext

8.3 Encryption

In order to encrypt a Plaintext into a Ciphertext, create a CryptoContext and a Plaintext as illustrated above.

The code in listing 15 illustrates this. It assumes that you have created a CryptoContext named "cc" and have created a Plaintext named ptxt within cc.

Observe that the code uses the KeyGen method to create a key pair, and uses the publicKey portion of that key pair for the encryption. The encryption will fail if the keys were generated in a different CryptoContext than cc.

```
////////////////////////////////////////////////////////
// Perform Key Generation Operation
////////////////////////////////////////////////////////

LPKeyPair<Poly> keyPair = cc->KeyGen();

if( !keyPair.good() ) {
        cout << "Key generation failed!" << endl;
        exit(1);
}

////////////////////////////////////////////////////////
// Encryption
////////////////////////////////////////////////////////

Ciphertext<Poly> ciphertext;

ciphertext = cc->Encrypt(keyPair.publicKey, ptxt);
```

Listing 15: Encrypting

8.4 Decryption

In order to decrypt a Ciphertext into a Plaintext, a CryptoContext, Ciphertext and KeyPair must exist. The keys and the Ciphertext must have been created within the CryptoContext. An example is shown in listing 16.

```
////////////////////////////////////////////////////////
//Decryption of Ciphertext
////////////////////////////////////////////////////////

Plaintext decrypted;

cc->Decrypt(keyPair.secretKey, ciphertext, &decrypted);
```

Listing 16: Decrypting

8.5 Re-Encryption

Re-Encryption involves converting a Ciphertext that is decryptable with key "A" into a Ciphertext that is decryptable with key "B" by creating a re-encryption key and using it to re-encrypt the ciphertext.

The code sample in listing 17 illustrates this capability. It assumes a CryptoContext cc, an initial keypair A, and a Ciphertext cipher that has been created using A. We show the generation of key pair B, the generation of the re-encryption key, the re-encryption operation, and the subsequent decryption of the re-encrypted Ciphertext with B's secret key.

Two variants of re-encryptions are shown. The first one corresponds to the Chosen Plaintext Attack (CPA) model. The second variant corresponds to the model secure under Honest Re-encryption Attacks (HRA) [Coh17]. We recommend the HRA model for practical use.

Note that in order to use Re-Encryption, the PRE capability must be enabled for the CryptoContext.

```
////////////////////////////////////////////////////////
// Perform Key Generation Operation
////////////////////////////////////////////////////////

// Initialize Key Pair Containers
LPKeyPair<Poly> B = cc->KeyGen();

if( !B.good() ) {
        cout << "Key generation failed!" << endl;
        exit(1);
}

////////////////////////////////////////////////////////
//Perform Re-encryption key generation operation.
////////////////////////////////////////////////////////

LPEvalKey<Poly> rekey =
        cc->ReKeyGen(B.publicKey, A.secretKey);

////////////////////////////////////////////////////////
// Re-Encryption
// CPA-secure variant
////////////////////////////////////////////////////////

auto re_cipher = cc->ReEncrypt(rekey, cipher);

////////////////////////////////////////////////////////
//Decryption of Ciphertext
////////////////////////////////////////////////////////

////////////////////////////////////////////////////////
// Re-Encryption with Ciphertext Re-randomization
// HRA-secure variant
////////////////////////////////////////////////////////

auto re_cipherHRA = cc->ReEncrypt(rekey, cipher,A.publicKey);

////////////////////////////////////////////////////////
//Decryption of Ciphertext
////////////////////////////////////////////////////////

Plaintext ptxtHRA;

cc->Decrypt(B.secretKey, re_cipherHRA, &ptxtHRA);
```

Listing 17: Re-Encrypting

8.6 Serialization and Deserialization

When used in client/server and other distributed application settings, PALISADE objects need to be saved to persistent storage or streams, and then loaded back to instantiate the PALISADE objects in memory. PALISADE provides a serialization capability based on the industry-standard C++ **cereal** library (see https://github.com/USCiLab/cereal). The objects can be serialized either as binary objects (fastest option) or as JSON files (in cross-platform scenarios).

PALISADE objects can be serialized to streams (**Serialize**/**Deserialize** methods) or files (**SerializeToFile**/**DeserializeFromFile** methods).

Many of the objects to be serialized are associated with a particular CryptoContext. In order for such objects to be correctly serialized and deserialized, the serialization saves the CryptoContext that the object belongs to as part of the serialization, and the deserialization makes sure that the CryptoContext for the object being deserialized matches the CryptoContext in the serialization.

A simple of example of serialization to files is provided in **src/pke/examples/simple-integers-serial.cpp**. A simple example of serialization to streams is provided in **pke/unittest/UnitTestSer.h**.

8.7 Homomorphic Addition of Ciphertexts

Homomorphic addition of two **Ciphertext** is performed by invoking the EvalAdd method of the **CryptoContext**. Both of the operands to the EvalAdd must have been created in the same **CryptoContext**, encrypted with the same **Key**, and encoded in the same way for this operation to work.

The code sample in listing 18 illustrates how to encode two vectors of integers into **Plaintext**, encrypt them into **Ciphertext**, perform homomorphic addition, and decrypt the result back into a **Plaintext**.

This code assumes a **CryptoContext** named cc and a keypair named kp.

```
// Encode source data

std::vector<int64_t> v1 = {3,2,1,3,2,1,0,0,0,0,0,0};
std::vector<int64_t> v2 = {2,0,0,0,0,0,0,0,0,0,0,0};
std::vector<int64_t> v3 = {1,0,0,0,0,0,0,0,0,0,0,0};
Plaintext p1 = cc->MakeCoefPackedPlaintext(v1);
Plaintext p2 = cc->MakeCoefPackedPlaintext(v2);
Plaintext p3 = cc->MakeCoefPackedPlaintext(v3);

// Encryption

auto c1 = cc->Encrypt(kp.publicKey, p1);
auto c2 = cc->Encrypt(kp.publicKey, p2);
auto c3 = cc->Encrypt(kp.publicKey, p3);

// EvalAdd Operation

auto c12 = cc->EvalAdd(c1,c2);
auto csum = cc->EvalAdd(c12,c3);

//Decryption after Accumulation Operation

Plaintext plaintextAdd;

cc->Decrypt(kp.secretKey, csum, &plaintextAdd);

plaintextAdd->SetLength(p1->GetLength());

cout << "Original Plaintext:" << endl;
cout << p1 << endl;
cout << p2 << endl;
cout << p3 << endl;

cout << "Resulting Added Plaintext:" << endl;
cout << plaintextAdd << endl;
```

Listing 18: Adding Ciphertexts

8.8 Homomorphic Multiplication of Ciphertexts

Homomorphic multiplication of two **Ciphertext** is performed by invoking the EvalMult method of the **CryptoContext**. Both of the operands to the EvalMult must have been created in the same **CryptoContext**, encrypted with the same **Key**, and encoded in the same way for this operation to work.

The code sample in listing 19 illustrates how to encode two vectors of integers into **Plaintext**, encrypt them into **Ciphertext**, perform homomorphic multiplication, and decrypt the result back into a **Plaintext**.

This code assumes a **CryptoContext** named cc and a keypair named kp.

Note the creation of an EvalMult key using EvalMultKeyGen at the beginning of the sample.

```
cc->EvalMultKeyGen(kp.secretKey);

// Encode source data

std::vector<int64_t> v1 = {3,2,1,3,2,1,0,0,0,0,0,0};
std::vector<int64_t> v2 = {2,0,0,0,0,0,0,0,0,0,0,0};
std::vector<int64_t> v3 = {1,0,0,0,0,0,0,0,0,0,0,0};
Plaintext p1 = cc->MakeCoefPackedPlaintext(v1);
Plaintext p2 = cc->MakeCoefPackedPlaintext(v2);
Plaintext p3 = cc->MakeCoefPackedPlaintext(v3);

// Encryption

auto c1 = cc->Encrypt(kp.publicKey, p1);
auto c2 = cc->Encrypt(kp.publicKey, p2);
auto c3 = cc->Encrypt(kp.publicKey, p2);

// EvalMult Operation

auto m12 = cc->EvalMult(c1,c2);
auto cprod = cc->EvalMult(m12,c3);

//Decryption after Multiplication Operation

Plaintext plaintextMul;

cc->Decrypt(kp.secretKey, cprod, &plaintextMul);

plaintextMul->SetLength(p1->GetLength());

cout << "Original Plaintext:" << endl;
cout << p1 << endl;
cout << p2 << endl;
cout << p3 << endl;

cout << "Resulting Plaintext:" << endl;
cout << plaintextMul << endl;
```
Listing 19: Multiplying Ciphertexts

9 Sample Implementations: FHE

PALISADE implements several fully homomorphic encryption schemes. Different schemes work with different input data types, and this is why this Section is organized based on three data types: integer numbers, real numbers, and binary bits (i.e., Boolean circuits). Each subsection describes which cryptographic schemes are appropriate for the respective data type, and how to start using the schemes in PALISADE.

9.1 Integer arithmetic

PALISADE provides two schemes that work with integer numbers, namely BFV and BGV. PALISADE includes RNS variants for both schemes. Here, we discuss BFV as an example.

In order to use BFV, users need to create a **CryptoContext** through the static factory method called **genCryptoContextBFV**. Among other arguments, this factory method takes the plaintext modulus, the security level, and the additive, multiplicative, and key-switching depths of the computation. The plaintext modulus determines an upper bound for the integer inputs in the BFV scheme. The security level is a parameter whose possible values correspond to 128-, 192-, or 256-bit security; PALISADE picks the appropriate underlying security parameters by consulting the relevant tables of the current FHE standard. Depending on the computation the user wants to carry out, she can set either the additive, multiplicative, or key-switching depth, and set all others to 0. Users who want to start using BFV can try running the examples at the following locations:

1. **src/pke/examples/simple-integers.cpp** - simple example showing how to instantiate the BFV scheme, generate keys, encrypt the data, perform homomorphic operations, and decrypt the result.
2. **src/pke/examples/simple-integers-serial.cpp** - shows how to serialize/deserialize objects when using the BFV scheme.

9.2 Real-number arithmetic

PALISADE supports real- and complex-number arithmetic through the use of the CKKS scheme. CKKS is implemented in RNS for better performance, so its **CryptoContext** can only be created with **DCRTPoly** as the template argument.

Creating a **CryptoContext** is done by an invocation of the static factory method **genCryptoContextCKKS**. Listing 20 shows all different arguments for **genCryptoContextCKKS**.

```
static CryptoContext<Element> genCryptoContextCKKS(
            usint multiplicativeDepth,
            usint scalingFactorBits,
            usint batchSize,
            SecurityLevel stdLevel = HEStd_128_classic,
            usint ringDim = 0,
            enum RescalingTechnique rsTech = EXACTRESCALE,
            enum KeySwitchTechnique ksTech = HYBRID,
            uint32_t numLargeDigits = 0,
            int maxDepth = 1,
            usint firstModSize = 60,
            usint relinWindow = 0,
            MODE mode = OPTIMIZED);
```

Listing 20: List of arguments of CKKS context generation method.

The CKKS scheme supports three different key switching algorithms, specifically digit decomposition by Brakerski and Vaikuntanathan (**BV**) [BV14], ciphertext modulus doubling by Gentry, Halevi and Smart (**GHS**) [GHS12], and the hybrid technique (**HYBRID**) [HK19] which combines ideas from the other two. Users can choose which algorithm the CKKS scheme will use by specifying the corresponding argument when creating the **CryptoContext**.

Besides choosing key switching algorithms, users can also choose between three variants called **EXACTRESCALE**, **APPROXRESCALE**, and **APPROXAUTO**. The former is simplest to use because it performs automatic rescaling, and introduces smaller rescaling error to computations of high multiplicative depth. This comes at a moderate performance cost of about 5-30% compared to **APPROXRESCALE**, which requires the user to track the depth of different ciphertexts and call **Rescale()** appropriately. The **APPROXAUTO** mode has the same rescaling approximation error as **APPROXRESCALE** but does the rescaling automatically.

CKKS provides proxy re-encryption and multiparty capabilities, but for PRE only the **BV** key switching algorithm is available. The multiparty capability supports all three key switching techniques.

Because of the way CKKS works, inputs have to be complex double numbers. CKKS also supports SIMD-like operations on packed data. Listing 21 shows how to create CKKS plaintexts. Users can also specify the depth and level of the plaintexts they want to create, by supplying these arguments to **MakeCKKSPackedPlaintext()**.

```
vector<complex<double>> x2 = { 5.0, 4.0, 3.0, 2.0, 1.0, 0.75, 0.5, 0.25 };

Plaintext ptxt1 = cc->MakeCKKSPackedPlaintext(x1);
```

Listing 21: Creating a CKKS plaintext.

The best way to start working with encrypted real numbers is to read the source code and related comments of the following examples:

1. **src/pke/examples/simple-real-numbers.cpp** - Simple example showing how to instantiate the CKKS scheme and perform basic homomorphic operations (addition, subtraction, multiplication, rotation).
2. **src/pke/examples/advanced-real-numbers.cpp** - Advanced examples illustrating usage of **HYBRID** key switching, **EXACTRESCALE** versus **APPROXRESCALE**, and hoisted rotations.

9.3 Boolean Circuit Arithmetic

PALISADE provides an implementation of FHEW to evaluate homorphically arbitrary Boolean circuits. Currently, PALISADE supports only the symmetric key encryption. The binary gates that are currently supported are OR, AND, NOR, NAND, XOR, and XNOR. All these gates take the same time, and internally call bootstrapping. The unary gate NOT is also supported; it's very fast (negligibly fast as compared to binary gate operations) and does not require any bootstrapping.

The following code samples are provided in the examples folder of the **binfhe** module:

1. **src/binfhe/examples/boolean.cpp** - simple example showing how to instantiate a **BinFHEContext**, generate keys, encrypt the data, evaluate a Boolean circuit, and decrypt the result.
2. **src/binfhe/examples/boolean-serial-json.cpp** - shows how to serialize/deserialize a crypto context, ciphertext, and keys in JSON representation. This representation is about one order of magnitude slower than the binary representation, and is typically useful only during debugging (or in special cross-platform scenarios).
3. **src/binfhe/examples/boolean-serial-binary.cpp** - shows how to serialize/deserialize a crypto context, ciphertext, and keys in the binary representation. This representation is recommeded for most practical scenarios.

10 Sample Implementations: ABE

10.1 Creating an ABEContext

An **ABEContext** can be created by either choosing the relevant security level or basic parameters related to security level.

```
/// Example showing creating an CP-ABE ABEContext
/// By specifying security level and number of attributes only
/// Element is NativePoly
/// First parameter is for security level, next parameter is the number of
   attributes

 ABEContext<NativePoly> context;
 context.GenerateCPABEContext(HEStd_192_classic,6);
```

Listing 22: Creating a ABEContext with Security Level

```
/// Example showing creating a GPV IBE ABEContext
/// By specifying relevant parameters: ring size and base of the gadget matrix
/// Element is NativePoly
/// First parameter is for ring size, second one is for base

 ABEContext<NativePoly> context;
 context.GenerateIBEContext(1024,64);
```

Listing 23: Creating a ABEContext with Parameters

10.2 Generating Master Keys

```
/// Example showing master key pair generation for GPV IBE ABEContext
/// Element is NativePoly

IBEMasterPublicKey<NativePoly> mpk;
IBEMasterSecretKey<NativePoly> msk;
context.Setup(&mpk,&msk);
```

Listing 24: Generating Master Public Key Pair with ABEContext

10.3 Creating Access Policies

```
/// Example showing user identifier generation for GPV IBE ABEContext
/// Element is NativePoly

IBEUserIdentifier<NativePoly> id(context.GenerateRandomElement());
```

Listing 25: Creating a User Identifier with GPV IBE ABEContext

```
/// Example showing user attribute set/access policy creating for CP-ABE
    ABEContext
/// Element is NativePoly

std::vector<unsigned int> userattributes = {1,0,1,0,1,1};
std::vector <int> accesspolicy = {0,-1,1,0,1,0};

CPABEUserAccess<NativePoly> ua(userattributes);
CPABEAccessPolicy<NativePoly> ap(accesspolicy);
```

Listing 26: Creating a User Attribute Set/Access Policy with CP-ABE ABEContext

10.4 Key Generation

The secret key generated is tailored to the identifier or attribute set of the user

```
/// Example showing secret key generation for GPV IBE ABEContext
/// Element is NativePoly
/// msk is the master secret key, mpk is the master public key, id is the user
     identifier

IBESecretKey<NativePoly> sk;
context.KeyGen(msk,mpk,id,&sk);
```

Listing 27: Generating a Secret Key with ABEContext

Additionally, it is possible to split the key generation process into two phases: Online and offline. Offline phase consists of creation of perturbation vector which is independent from the user's id or attribute set.

```
/// Example showing the offline phase of secret key generation for GPV IBE
    ABEContext
/// Element is NativePoly
/// msk is the master secret key

PerturbationVector<NativePoly> pv;
context.KeyGenOfflinePhase(msk,pv);
```

Listing 28: Generating a Secret Key with ABEContext (Offline Phase - Perturbation Generation)

The online phase in this scenario consists of actual key generation based on user's id or attribute set without the perturbation subroutine.

```
/// Example showing the online phase of secret key generation for GPV IBE
    ABEContext
/// Element is NativePoly
/// msk is the master secret key, mpk is the master public key, id is the user
    's identifier and pv is the perturbation vector

IBESecretKey<NativePoly> sk;
context.KeyGenOnlinePhase(msk,mpk,id,pv,&sk);
```

Listing 29: Generating a Secret Key with ABEContext (Online Phase - Key Generation)

10.5 Encryption

A **Plaintext** is encrypted with a target party, which means the target user's identifier in GPV IBE or access policy for CP-ABE.

```
/// Example showing encryption for CP-ABE ABEContext
/// Element is NativePoly
///mpk is the master public key, ap is the access policy

///Creation of plaintext. The format is intentionally used for support
 std::vector<int64_t> vectorOfInts = { 1,0,0,1,1,0,1,0, 1, 0};
 Plaintext pt = context.MakeCoefPackedPlaintext(vectorOfInts);

///Actual encryption
 CPABECiphertext<NativePoly> ct;
context.Encrypt(mpk,ap,pt,&ct);
```

Listing 30: Encryption with CP-ABE ABEContext

10.6 Decryption

```
/// Example showing decryption for CP-ABE ABEContext
/// Element is NativePoly
///sk is the secret key of the user, ua is the user's attributes, ap is the
    access policy defined and ct is the ciphertext

Plaintext dt = context.Decrypt(ap,ua,sk,ct);
```

Listing 31: Decryption with CP-ABE ABEContext

```
/// Example showing decryption for GPV IBE ABEContext
/// Element is NativePoly
///sk is the secret key of the user and ct is the ciphertext

Plaintext dt = context.Decrypt(sk,ct);
```

Listing 32: Decryption with GPV IBE ABEContext

11 Sample Implementations: Digital Signature

11.1 Creating a SignatureContext

A **SignatureContext** can be created by either choosing one of the predefined ring sizes or a set of parameters.

```
/// Example showing creating a GPV SignatureContext
/// By specifying some specific parameters
/// First parameter is the ring size, second parameter is the bit width of
    modulus and third one is the base of the gadget matrix
/// Element is NativePoly

SignatureContext<NativePoly> context;
context.GenerateGPVContext(1024,61,64);
```

Listing 33: Creating a SignatureContext with Parameters

11.2 Key Generation

```
/// Example showing creating sign/verification keys  with a GPV
    SignatureContext
/// Element is NativePoly

GPVVerificationKey<NativePoly> vk;
GPVSignKey<NativePoly> sk;
context.KeyGen(&sk,&vk);
```

Listing 34: Key Generation with SignatureContext

11.3 Signing

```
/// Example showing signing  with a GPV SignatureContext
/// Element is NativePoly
///sk is the signing key and vk is the verification key

///Creation of plaintext
string pt = "This is a test";
GPVPlaintext<NativePoly> plaintext(pt);

///Actual signing
GPVSignature<NativePoly> signature;
context.Sign(plaintext,sk,vk,&signature);
```

Listing 35: Signing with SignatureContext

Additionally, it is possible to split the signing process into two phases: Online and offline. Offline phase consists of creation of perturbation vector which is independent from message to be signed.

```
/// Example showing offline phase of signing  with a GPV SignatureContext
/// Element is NativePoly
///sk is the signing key

PerturbationVector<NativePoly> pv;
context.SignOfflinePhase(sk,pv);
```

Listing 36: Signing with SignatureContext (Offline Phase - Perturbation Generation)

The online phase in this scenario consists of actual signing process without the perturbation subroutine.

```
/// Example showing online phase of signing  with a GPV SignatureContext
/// Element is NativePoly
///sk is the signing key, vk is the verification key and pv is the
    perturbation vector generated in offline phase

///Creation of plaintext
string pt = "This is a test";
GPVPlaintext<NativePoly> plaintext(pt);

///Actual signing
GPVSignature<NativePoly> signature;
context.SignOnlinePhase(plaintext,sk,vk,pv,&signature);
```

Listing 37: Signing with SignatureContext (Online Phase - Signing)

11.4 Verification

```
/// Example showing verification  with a GPV SignatureContext
/// Element is NativePoly
/// vk is the verification key

bool verificationResult = context.Verify(plaintext,signature,vk);
```

Listing 38: Verification with SignatureContext

12 Building and Installing PALISADE

PALISADE git repository includes an up-to-date wiki with detailed instructions on building and running PALISADE on various platforms. We use CMake to build PALISADE, and support Linux, Windows, and macOS. The library requires a C++ compiler that implements the C++11 standard, with support for the OpenMP library. Make, cmake, and autoconf also need to be installed. While not required, it is also recommended that you install and use doxygen.

13 Programming Style

PALISADE coding style is based on the official Google C++ coding style.

Of particular note on the documentation style:

- We use doxygen commenting style on classes, methods and constants.
- We given meaningful variable names to all variables.
- Every reused discrete block of code has its own method.
- Every discrete line or code or discrete group of code lines for each task has its own comment.

With regards to naming conventions:

- Variable names: camelCase.
- Class, struct, typedef, and enum names: CamelCase.
- Class data members: m_camelCase.
- Class accessor names: GetProperty() and SetProperty().
- Class method names: CamelCase.
- Global variable names: g_camelCase.
- Constant names and macros: UPPER_CASE_WITH_UNDERSCORES (example: BIT_LENGTH).
- Operator overloading is used for binary operations.

We also follow the additional design principles that:

- cout should never be used for exception handling and should never be used in committed code in the core PALISADE library.
- a set of PALISADE exceptions is defined in utils/exception.h. The library is being migrated to throw only these exceptions.

A PALISADE License

PALISADE is available under the following license:

B Support

The following organizations and projects have been providing generous sponsorship for the PALISADE project:

- DARPA SAFEWARE (2015-present) : Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Army Research Laboratory (ARL) under Contract Numbers W911NF-15-C-0226 and W911NF-15-C-0233. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the Department of Defense or the U.S. Government.
- DARPA MARSHAL (2017-2019): Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the US Navy SPAWAR Systems Center Pacific (SSCPAC) under Contract Number N66001-17-1-4043. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the Department of Defense or the U.S. Government.
- NSA CAE (2015-2016) : Project sponsored by the National Security Agency under Grant H98230-15-1-0274. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein.
- IARPA RAMPARTS (2016-2018) : This research is based upon work supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either express or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.
- NIH SBIR GEARS (2017-2018): Research reported in this publication was supported by National Human Genome Research Institute of the National Institutes of Health under award number 1R43HG010123.
- Simons Foundation (2015-2016)
- Sloan Foundation (2017-present)

References

[BEHZ16] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[BGP+19] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, Kurt Rohloff, and Vinod Vaikuntanathan. Optimized homomorphic encryption solution for secure genome-wide association studies. Cryptology ePrint Archive, Report 2019/223, 2019. `https://eprint.iacr.org/2019/223`.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[BPA+18] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme. Cryptology ePrint Archive, Report 2018/589, 2018. `https://eprint.iacr.org/2018/589`.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology–CRYPTO 2012*, pages 868–886. Springer, 2012.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[CGGI18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *IACR Cryptology ePrint Archive*, 2018:421, 2018.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/931, 2018. `https://eprint.iacr.org/2018/931`.

[CKKS16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Report 2016/421, 2016. `https://eprint.iacr.org/2016/421`.

[Coh17] Aloni Cohen. What about bob? the inadequacy of cpa security for proxy reencryption. Cryptology ePrint Archive, Report 2017/785, 2017. `https://eprint.iacr.org/2017/785`.

[DM14] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Report 2014/816, 2014. `https://eprint.iacr.org/2014/816`.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[GINX16] Nicolas Gama, Malika Izabachène, Phong Q. Nguyen, and Xiang Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 528–558, 2016.

[GPR+17] Kamil Doruk Gür, Yuriy Polyakov, Kurt Rohloff, Gerard W. Ryan, Hadi Sajjadpour, and Erkay Savaş. Practical applications of improved gaussian sampling for trapdoor lattices. Cryptology ePrint Archive, Report 2017/1254, 2017. `https://eprint.iacr.org/2017/1254`.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, pages 197–206, 2008.

[HK19] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. Cryptology ePrint Archive, Report 2019/688, 2019. `https://eprint.iacr.org/2019/688`.

[HPS18] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. 2018. `https://eprint.iacr.org/2018/117`.

[KPP20] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. A more usable RNS variant of CKKS homomorphic encryption scheme, 2020. in preparation.

[KPZ20] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. A more usable RNS variant of BGV homomorphic encryption scheme, 2020. in preparation.

[LN14] Tancrede Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. In *International Conference on Cryptology in Africa*, pages 318–335. Springer, 2014.

[MP20] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in fhew-like cryptosystems. Cryptology ePrint Archive, Report 2020/086, 2020. `https://eprint.iacr.org/2020/086`.

[SS11] Damien Stehlé and Ron Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. In KennethG. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer Berlin Heidelberg, 2011.

[ZZG12] Jiang Zhang, Zhenfeng Zhang, and Aijun Ge. Ciphertext policy attribute-based encryption from lattices. In Heung Youl Youm and Yoojae Won, editors, *7th ACM*

Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012, pages 16–17. ACM, 2012.