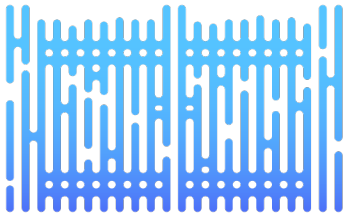


PALISADE

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

March 26, 2021

Yuriy Polyakov

contact@palisade-crypto.org

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS

- Tutorial with applications consisting of 4 episodes (6 lectures)
- Episode 1
  - Introduction to Homomorphic Encryption
  - Boolean Arithmetic with Applications
- Episode 2
  - Integer Arithmetic
  - Applications of Homomorphic Encryption over Integers
- Episode 3
  - Introduction to Multiparty Homomorphic Encryption
- **Episode 4**
  - **Introduction to Approximate Homomorphic Encryption**

PALISADE

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

Introduction to Approximate Homomorphic Encryption

Yuriy Polyakov
ypolyakov@dualitytech.com

PREREQUISITES FOR THIS TALK

- Webinar #2A: Introduction to Homomorphic Encryption (https://www.youtube.com/watch?v=rMDoZdH53ZM)
- Other related recorded webinars can be accessed at https://palisade-crypto.org/webinars/

AGENDA

- Basics
  - Motivation for approximate homomorphic encryption
  - Fixed-point approximate arithmetic with rescaling
  - How is data represented?
  - Complete list of primitive operations
- Parameter selection
  - Scaling factor
  - Ciphertext modulus / Multiplicative depth
  - Ciphertext dimension / Security level
- Simple code example
- Advanced topics
  - Selected high-level CKKS operations
  - Rescaling modes
  - Key switching modes
  - Multiparty CKKS
  - Security of CKKS for the scenario of shared decryptions
  - High-precision CKKS
- Advanced examples
  - Advanced code examples included with PALISADE
  - Open-source implementations using CKKS in PALISADE

Basics

Explains motivation for approximate HE, supported operations and data encoding

MOTIVATION FOR APPROXIMATE HE

- Good for any application where we work with real numbers, e.g., where we use floating-point numbers
  - Encrypt real numbers and perform addition and multiplication with the result rounded to a fixed precision, for instance, two digits after the decimal point
    - 12.42 + 1.34 = 13.76, 2.23 * 5.19 = 11.57
- The main limitation of integer homomorphic encryption (BGV/BFV) is that it requires very large plaintext moduli to support operations over real numbers
  - All computations in integer HE are performed exactly
  - Integer HE rapidly becomes inefficient as further multiplications are performed
- Approximate homomorphic encryption allows dropping least significant bits by *rescaling* the encrypted data, similar to how it is done for floating-point numbers in practice
- Typical applications of approximate HE
  - Statistical computations
  - Polynomial evaluation
  - Matrix arithmetic
  - Regression inference and training
  - Evaluation of non-linear/non-smooth functions using their polynomial approximations

FIXED-POINT ARITHMETIC WITH RESCALING

- Suppose we have two numbers: a = 0.125654 and b = 3.534365
- We choose a scaling factor Δ that converts these real numbers into integers (like in fixed-point arithmetic)
  - In this case, $\Delta = 10^6$ is a good choice to maintain the same precision
- We multiply a and b by the scaling factor Δ, yielding a' = a * Δ = 125654 and b' = b * Δ = 3534365
- If we want to compute $a + b$, we do the following
  - $a' + b'$ = 3660019, which corresponds to $a + b = \frac{a' + b'}{\Delta}$ = 3.660019
- If we want to compute $a * b$, we do the following
  - $a' * b'$ = 444107099710, which corresponds to $a * b = \frac{a' * b'}{\Delta^2}$ = 0. 444107099710
  - We convert $a * b$ to the original scale Δ by dropping the 6 least significant digits (dividing by Δ), i.e.,
    - Compute $\frac{a' * b'}{\Delta}$ = 444107 and use it for future computations
  - Scaling down by Δ is called *rescaling*

FIXED-POINT ARITHMETIC WITH RESCALING IN CKKS

- The Cheon-Kim-Kim-Song (CKKS) scheme is the approximate homomorphic encryption implemented in PALISADE [CKKS17]
  - Multiple variants of the CKKS are implemented in PALISADE, but they all share common properties and vary only in the approximation error, performance, and usability
- The fixed-point arithmetic with rescaling in CKKS has additional two features
  - Binary fixed-point arithmetic, i.e., $\Delta = 2^p$ is used instead of decimal fixed-point arithmetic
  - CKKS operations introduce extra approximation error, which "erases" typically 12-25 least significant bits
    - The actual scaling factor in CKKS should be set to a higher value to compensate for the CKKS approximation error
- By default, the CKKS implementation in PALISADE automatically performs rescaling, similar to how normalization and other internal adjustments are done in double-precision floating-point arithmetic

MAIN DATA STRUCTURE

- The main data structure is a vector (array) of real numbers
- Many real numbers (typically between 2K and 64K) are “packed” in one vector (ciphertext)
  - Let us denote the vector size as *n* (a power of two)
- Addition and multiplication of *n* real numbers can be done using a single addition/multiplication
  - Similar to Single Instruction Multiple Data (SIMD) instruction sets available on many modern processors
  - The SIMD capability should be used as much as possible to achieve best efficiency
- **Rotation** operation is added to allow accessing the value at a specific index of the array
- Addition, multiplication, and rotation are three primitive operations in approximate HE

COMPLETE LIST OF PRIMITIVE OPERATIONS

- Two-argument operations (the plaintext can represent a vector of real numbers or a single real number)
  - Ciphertext-Ciphertext addition: **EvalAdd**
  - Ciphertext-Plaintext addition: **EvalAdd**
  - Ciphertext-Ciphertext multiplication: **EvalMult**
  - Ciphertext-Plaintext multiplication: **EvalMult**
  - Ciphertext-Ciphertext subtraction: **EvalSub**
  - Ciphertext-Plaintext subtraction: **EvalSub**
- Unary operations
  - Negation: **EvalNegate**
  - Vector rotation: **EvalAtIndex**
- The result of all these operations is a ciphertext, i.e., an encrypted vector
  - The benefit of this in practice is that mixed model-data modes can be supported, e.g.,
    - Encrypted model, data in the clear
    - Model in the clear, encrypted data

DATA ENCODING

- Packing technique: **CKKSPackedEncoding**
  - Packs real numbers into a vector of size *n*
  - Supports component-wise addition (**EvalAdd**) and multiplication (**EvalMult**)

```
[ 1.1 ]   [ 4.4 ]   [ 5.5 ]   [ 1.5 ]   [ 4.5 ]   [  6.75 ]
[ 2.2 ] + [ 5.5 ] = [ 7.7 ],  [ 2.5 ] * [ 5.0 ] = [ 12.50 ]
[ 3.3 ]   [ 6.6 ]   [ 9.9 ]   [ 3.5 ]   [ 6.1 ]   [ 21.35 ]
```

  - Adds a new rotation operation (**EvalAtIndex**)
    - Right shift: positive index
    - Left shift: negative index
    - Rotations work cyclically over a vector of size *n*
  - Always used

Parameter selection

Explains main parameters and provides recommendations for their selection

MAIN PARAMETERS

- Scaling factor $\Delta = 2^p$
  - Determines the precision of computations
- Ciphertext modulus q
  - Functional parameter that determines how many computations are allowed (how much noise can be tolerated)
  - Often set implicitly using the value of multiplicative depth specified by the user
- Ciphertext dimension N
  - Minimum value is computed based on the desired security level and ciphertext modulus q
  - It is also double the size of the vector of encrypted real numbers, i.e., $N = 2n$

GUIDELINES FOR SETTING SCALING FACTOR

- The scaling factor $\Delta = 2^p$ determines the precision after the radix point
  - Think of this precision in the fixed-point sense
- CKKS "erases" 12-25 least significant bits (depending on the multiplicative depth)
  - Each addition and multiplication may consume up to 1 bit (typically significantly less)
- Hence the value of *p* for desired precision *v* is something like $p \approx v + 20$
  - It can be adjusted as needed, depending on the multiplicative depth of the computation
  - Just like floating-point arithmetic, approximate homomorphic encryption introduces an error, and errors from prior computations get accumulated
- PALISADE outputs estimated precision for a decrypted CKKS result

GUIDELINES FOR SETTING CIPHERTEXT MODULUS

- Ciphertext modulus *q* is the main functional parameter that is determined by the computation
  - Each arithmetic operation increases the noise, and *q* should be large enough to accommodate the noise from all arithmetic operations
  - From the noise perspective, multiplication is much costlier than addition
  - In PALISADE, *q* is automatically computed based on the multiplicative depth and scaling factor Δ
- Multiplicative depth is not necessarily the number of multiplications
  - For example, if we need to compute **a*b*c*d**, we can compute **e=a*b** and **f=c*d** using one level, and then compute **e*f** using the second level. Hence we use 2 levels (depth of 2) rather 3 if we were to do the multiplication sequentially.
  - This technique is called **binary tree multiplication**, and it should be used to minimize the multiplicative depth wherever possible.

GUIDELINES FOR SETTING CIPHERTEXT DIMENSION

- Ciphertexts are represented as two arrays of size *N*
- This size *N*, called ciphertext dimension, should have a certain minimum value to comply with the chosen security level and desired ciphertext modulus
- Main options for security levels in PALISADE (we implemented the recommendations from the HE standard published at HomomorphicEncryption.org):
  - *HEStd_128_classic* – 128-bit security against classical computers
  - *HEStd_192_classic* – 192-bit security against classical computers
  - *HEStd_256_classic* – 256-bit security against classical computers
  - *HEStd_NotSet* – toy settings (for debugging and prototype development)
- The ciphertext dimension *N* also determines the size of the vector of encrypted real numbers ($n = N/2$).
  - It may sometimes be useful to use a larger ring dimension than the minimum one needed for security.
  - In this case, the user can specify the ring dimension explicitly.

Simple code example

Explains a simple example showing how to do additions, multiplications, and rotations in CKKS

KEY CONCEPTS/CLASSES

- **CryptoContext**
  - A wrapper that encapsulates the scheme, crypto parameters, encoding parameters, and keys
  - Provides the same API for all HE schemes
- **Ciphertext**
  - Stores the ciphertext polynomials
- **Plaintext**
  - Stores the plaintext data (both raw and encoded)
  - Supports only **CKKSPackedEncoding**

STEP 1 – SET CRYPTOCONTEXT

```
// Set the main parameters
uint32_t multDepth = 1;
uint32_t scaleFactorBits = 50;
uint32_t batchSize = 8;
SecurityLevel securityLevel = HEStd_128_classic;

// Instantiate the crypto context
CryptoContext<DCRTPoly> cc =
    CryptoContextFactory<DCRTPoly>::genCryptoContextCKKS(
        multDepth, scaleFactorBits, batchSize, securityLevel);

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

STEP 2 – KEY GENERATION

```
// Generate a public/private key pair
auto keys = cc->KeyGen();

// Generate the relinearization key
cc->EvalMultKeyGen(keys.secretKey);

// Generate the rotation evaluation keys
cc->EvalAtIndexKeyGen(keys.secretKey, {1, -2});
```

STEP 3 – ENCRYPTION

```
// Inputs
vector<double> x1 = {0.25, 0.5, 0.75, 1.0, 2.0, 3.0, 4.0, 5.0};
vector<double> x2 = {5.0, 4.0, 3.0, 2.0, 1.0, 0.75, 0.5, 0.25};

// Encoding as plaintexts
Plaintext ptxt1 = cc->MakeCKKSPackedPlaintext(x1);
Plaintext ptxt2 = cc->MakeCKKSPackedPlaintext(x2);

std::cout << "Input x1: " << ptxt1 << std::endl;
std::cout << "Input x2: " << ptxt2 << std::endl;

// Encrypt the encoded vectors
auto c1 = cc->Encrypt(keys.publicKey, ptxt1);
auto c2 = cc->Encrypt(keys.publicKey, ptxt2);
```

STEP 4 – EVALUATION

```
// Homomorphic addition
auto cAdd = cc->EvalAdd(c1, c2);

// Homomorphic subtraction
auto cSub = cc->EvalSub(c1, c2);

// Homomorphic scalar multiplication
auto cScalar = cc->EvalMult(c1, 4.0);

// Homomorphic multiplication
auto cMul = cc->EvalMult(c1, c2);

// Homomorphic rotations
auto cRot1 = cc->EvalAtIndex(c1, 1);
auto cRot2 = cc->EvalAtIndex(c1, -2);
```

STEP 5 – DECRYPTION

```
Plaintext result;

// Decrypt the result of addition
cc->Decrypt(keys.secretKey, cAdd, &result);
result->SetLength(batchSize);
std::cout << "x1 + x2 = " << result;
std::cout << "Estimated precision in bits: " << result->GetLogPrecision() << std::endl;

// Decrypt the result of subtraction
cc->Decrypt(keys.secretKey, cSub, &result);

// Decrypt the result of scalar multiplication
cc->Decrypt(keys.secretKey, cScalar, &result);

// Decrypt the result of multiplication
cc->Decrypt(keys.secretKey, cMul, &result);

// Decrypt the result of rotations
cc->Decrypt(keys.secretKey, cRot1, &result);
cc->Decrypt(keys.secretKey, cRot2, &result);
```

Advanced topics

Discusses some non-primitive operations available in PALISADE and other advanced topics

SELECTED HIGHER-LEVEL OPERATIONS

Operation	Input arguments	Description
EvalSum	ciphertext, *batchSize*	Computes a sum of *batchSize* components in an encrypted vector; if *batchSize* < *n*, the vector of size *batchSize* needs to be replicated *n*/*batchSize* times
EvalInnerProduct	2 ciphertexts, *batchSize*	Multiplies two vectors, and then computes **EvalSum**
EvalMultMany	*k* ciphertexts	Computes a product of *k* ciphertexts using the binary tree approach (only log *k* depth is needed)
EvalMerge	*k* ciphertexts	Merges *k* ciphertexts with encrypted results in first slot into a ciphertext with *k* slots
EvalFastRotation	ciphertext, *index, precomp.*	Fast rotation when multiple rotations of the same ciphertext are needed
EvalLinearWSum	*k* ciphertexts, *k* constants (weights)	Computes a linear weighted sum of *k* ciphertexts with *k* weights
EvalPoly	ciphertext, *k* coefficients	Evaluates a polynomial of degree *k* - 1

RESCALING MODES

Rescaling mode	How rescaling is done	Usability	Approximation error	Performance
EXACTRESCALE (default)	Automatically	Easy	Smallest	Slowest (by 1.2x – 1.5x vs. APPROXRESCALE)
APPROXAUTO	Automatically	Easy	Medium (few bits more)	Medium
APPROXRESCALE	Manually	Hard	Slightly higher than APPROXAUTO	Fastest if used by an HE expert

More details on rescaling modes:
A. Kim, A. Papadimitriou, and Y. Polyakov, Approximate Homomorphic Encryption with Reduced Approximation Error, Cryptology ePrint Archive, Report 2020/1118, https://eprint.iacr.org/2020/1118

KEY SWITCHING MODES

Mode	Number of digits	Relinearization window	Noise growth	Performance
HYBRID (default)	0 (default) typically 3 or mult. depth + 1	N/A	Small/almost negligible	Best for deeper computations, comparable to BV for shallow computations
GHS (to be deprecated)	N/A	N/A	Small/almost negligible	Worse than HYBRID
BV	N/A	0 (default) typically 20	High for rotations	Fastest for very shallow computations

More details on key switching modes can be found in the Appendix of
A. Kim, Y. Polyakov, and V. Zucca, Revisiting Homomorphic Encryption Schemes for Finite Fields, Cryptology ePrint Archive, Report 2021/204, https://eprint.iacr.org/2021/204

MULTIPARTY SUPPORT

- PALISADE CKKS implementation supports multiparty FHE using the threshold FHE approach
- For details, see PALISADE webinar #4: https://www.youtube.com/watch?v=9Fa6rFUyQ_w&t=3s

CKKS SECURITY FOR SCENARIO OF SHARED DECRYPTIONS

- Li and Micciancio recently showed (B. Li and D. Micciancio, On the Security of Homomorphic Encryption on Approximate Numbers, EUROCRYPT'21, https://eprint.iacr.org/2020/1533) that the IND-CPA security may not be strong enough when decryption results need to be published or shared with untrusted parties
- PALISADE implemented a mitigation for this scenario that modifies the decryption procedure by adding Gaussian noise proportional to current approximation noise (this mitigation is always applied)
- PALISADE also includes a compile-level Cmake parameter (CKKS_M_FACTOR) to increase the magnitude of added noise in scenarios where a very large number of decryption queries is allowed, effectively supporting the noise flooding conditions typically needed for provable security
- More details can be found at https://palisade-crypto.org/security-of-ckks/ and **Security.md** in the PALISADE code

HIGH-PRECISION CKKS

- Starting with v1.11 (scheduled to be released on March 31, 2021), PALISADE includes a high-precision CKKS implementation
  - Scaling factor in this case can be as large as 2^{119} (compared to 2^{59} for the regular CKKS implementation in PALISADE)
  - The high-precision CKKS implementation provides support for double-precision arithmetic, i.e., 52 bits if the scaling factor is set to 70 or more bits
  - Another benefit is the support for any practically reasonable value of CKKS_M_FACTOR when using CKKS in conditions where numerous decryption queries are allowed (strongest security model where CKKS may be used)
  - Runtime is about 4x slower than for regular CKKS implementation

Advanced examples

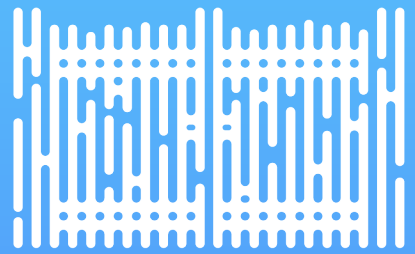
Describes more advanced examples in PALISADE and separate git repositories

ADVANCED CODE EXAMPLES

- **src/pke/examples/simple-real-examples-serial.cpp**
  - Simple real example with serialization of keys and ciphertexts
- **src/pke/examples/advanced-real-numbers.cpp**
  - Advanced example illustrating different rescaling modes, key switching modes, and fast rotations
- **src/pke/examples/advanced-real-numbers-128.cpp**
  - Equivalent of advanced-real-numbers.cpp for high-precision CKKS
- **src/pke/examples/polynomial_evaluation.cpp**
  - Examples of polynomial evaluation
- **src/pke/examples/threshold-fhe.cpp**
  - Example of multiparty CKKS

APPLICATION GIT REPOSITORIES USING CKKS IN PALISADE

- https://gitlab.com/duality-technologies-public/palisade-gwas-demos/
  - Secure large-scale genome-wide association studies
  - Results published in PNAS (2020): https://doi.org/10.1073/pnas.1918257117
- https://gitlab.com/palisade/palisade-python-demo
  - Linear Support Vector Machine classifier
- https://gitlab.com/palisade/palisade-serial-examples
  - CKKS example using serialization over sockets



THANK YOU

ypolyakov@dualitytech.com

https://palisade-crypto.org

PALISADE