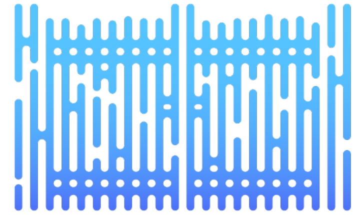


PALISADE

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

October 2, 2020

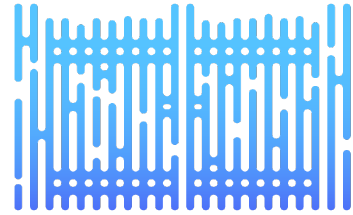
Yuriy Polyakov

David Bruce Cousins

contact@palisade-crypto.org

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS

- Tutorial with applications consisting of 3 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
  - Approximate Number Arithmetic
  - Applications of Homomorphic Encryption over Approximate Numbers



PALISADE

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

Integer Arithmetic

Yuriy Polyakov
ypolyakov@dualitytech.com

PREREQUISITES FOR THIS TALK

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

PALISADE

AGENDA

- Basics
  - What arithmetic operations are supported?
  - What data structures are used?
  - Complete list of primitive operations
- Data encoding techniques
- Parameter selection
  - Plaintext modulus
  - Ciphertext modulus / Multiplicative depth
  - Ciphertext dimension / Security level
- Which scheme to choose?
- Code example
- More advanced topics
  - Higher-level operations
  - Further encoding-related topics

Explains supported operations and data structures

WHAT ARITHEMTIC OPERATIONS ARE SUPPORTED?

- Exact integer arithmetic
  - Encrypt small integers and perform **addition** and **multiplication**, as long as the result does not exceed some fixed bound, for instance, if the bound is 10000
    - 123 + 456 → 579
    - 12 * 432 → 5184
    - 35 * 537 → overflow
  - Most common scenario
  - In PALISADE, the maximum supported bound is 2^{32} (by design), i.e., the support is limited to 32-bit integers
- Modular integer arithmetic (finite fields)
  - Encrypt 8-bit unsigned integers (between 0 and 255) and perform **addition** and **multiplication** modulo 256
    - 128 + 128 → 0
    - 2 * 129 → 2
  - Used only for special use cases

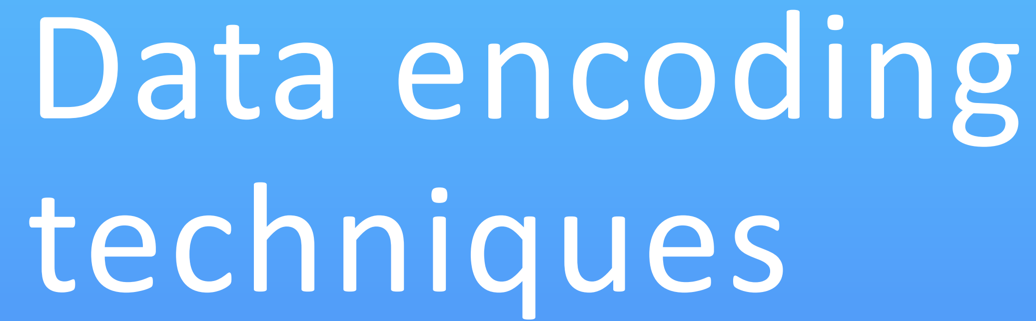
WHAT DATA STRUCTURES ARE USED?

- The main data structure is a vector (array) of bounded integers
  - Many integers (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 integers 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 integer HE
- More advanced structures are supported but they are used less frequently and typically require more advanced math knowledge, including
  - Matrices of bounded integers
  - Polynomials with bounded coefficients
  - High-precision integers (one per ciphertext)

PALISADE

COMPLETE LIST OF PRIMITIVE OPERATIONS

- Two-argument operations (the plaintext can represent a vector or a scalar)
  - 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



Introduces main data encoding techniques used in integer HE

MAIN DATA ENCODING TECHNIQUE

- Standard packing: **PackedEncoding**
  - Packs bounded integers into a vector of size *n*
  - Supports component-wise addition (**EvalAdd**) and multiplication (**EvalMult**)

```
[ 1 ]   [ 4 ]   [ 5 ]   [ 1 ]   [ 4 ]   [  4 ]
[ 2 ] + [ 5 ] = [ 7 ],  [ 2 ] * [ 5 ] = [ 10 ]
[ 3 ]   [ 6 ]   [ 9 ]   [ 3 ]   [ 6 ]   [ 18 ]
```

  - Adds a new rotation operation (**EvalAtIndex**)
    - Right shift: positive index
    - Left shift: negative index
    - Rotations work cyclically over two equal “subvectors” of size $n/2$
  - Used almost always

OTHER (RARELY USED) DATA ENCODING TECHNIQUES

- Coefficient packing: **CoefPackedEncoding**
  - Packs bounded integers into a vector of size *n*
  - Supports only component-wise addition (**EvalAdd**) but not multiplication
  - Scalar multiplication and limited rotation capability are also supported
  - Typically works well when no multiplications are needed
- Integer encoding: **IntegerEncoding**
  - Packs one integer into one ciphertext
  - Supports high-precision arithmetic but is not does not utilize packing (much slower)

Parameter selection

Explains main parameters and provides recommendations for their selection

MAIN PARAMETERS

- Plaintext modulus p
  - The bound for integer arithmetic
  - The modulus for modular arithmetic
- 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 the size of the vector of encrypted integers when standard or coefficient packing is used

GUIDELINES FOR SETTING PLAINTEXT MODULUS

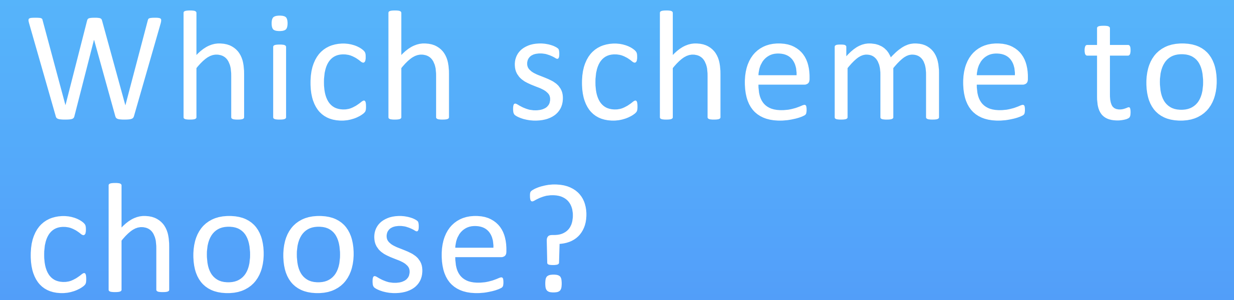
- In the case of exact integer arithmetic, the plaintext modulus *p* should be large enough to avoid an overflow
  - As we do not know the encrypted value, *p* should be estimated using the worst-case assumption
    - If we have two inputs *a* in [0,18] and *b* in [0,257] and we need to compute *a***b*, the value of *p* should be at least 18*257+1 = 4627
- If we use standard packing (**PackedEncoding**), we have to compute a special prime that is compatible with this encoding method
  - **auto plaintextModulus = FirstPrime<NativeInteger>(bits, 2*n);**
    - *bits* – the plaintext modulus should be at least 2^{bits} based on computation requirements
    - If *n* is not known (automatically computed), you can use n = 65,536
  - A convenient plaintext modulus for most cases: *p* = 65,537
- For all other encoding types, an arbitrary plaintext modulus can be used as long as it does not overflow in the case of exact integer arithmetic
- Overflow is not an issue for modular integer arithmetic

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 plaintext modulus p
- 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. So 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 integers.
  - 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.



Introduces BFV and BGV, and explains main differences between them

BFV and BGV schemes

- Brakerski/Fan-Vercauteren (BFV) scheme
  - Use the Most Significant Digit (MSD) form to encode messages
  - The ciphertext modulus is constant while the noise increases with every operation
  - Has an expensive homomorphic multiplication operation
  - Two roughly equivalent variants are implemented in PALISADE: **BFVrns** and **BFVrnsB** (a mixed multiprecison-RNS variant **BFV** has been supported since 2017 but it is not recommended anymore as it is much less efficient)
    - **BFVrns** is slightly faster and has been more exhaustively stress-tested in PALISADE
    - **BFVrns** was added in December 2017
    - **BFVrnsB** was added in June 2018
- Brakerski-Gentry-Vaikuntanathan (BGV) scheme
  - Use the Least Significant Digit (LSD) form to encode messages
  - Maintains the same level of noise by reducing the ciphertext modulus after each multiplication
  - Supports much faster homomorphic multiplication
  - Called **BGVrns** in PALISADE (a mixed multiprecision-RNS variant **BGV** has been supported since 2017 but it is not recommended anymore as it is much less efficient)
    - **BGVrns** was recently added in v1.10 (June 2020)

BGV and BFV schemes

- Notes on the current implementation in PALISADE
  - Theoretically speaking, BGV and BFV have roughly the same noise growth
  - But the current BGV implementation in PALISADE does not yet select the most efficient parameters by default
    - Most efficient parameters can be set manually but require FHE expertise
  - Some further improvements to BGV will be added in the next version of PALISADE
- Recommendations
  - For production-like deployments, **BFVrns** is recommended
  - **BGVrns** may give better performance for many computations, especially where many homomorphic multiplications are performed
    - This implementation can be used in research projects

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

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 multiple encodings in a polymorphic manner, including **PackedEncoding**, **IntegerEncodering**, **CoefPackedEncoding**.

STEP 1 – SET CRYPTOCONTEXT

```
// Set the main parameters
int plaintextModulus = 65537;
double sigma = 3.2;
SecurityLevel securityLevel = HEStd_128_classic;
uint32_t depth = 2;

// Instantiate the crypto context
CryptoContext<DCRTPoly> cryptoContext =
    CryptoContextFactory<DCRTPoly>::genCryptoContextBFVrns(
        plaintextModulus, securityLevel, sigma, 0, depth, 0, OPTIMIZED);

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

STEP 2 – KEY GENERATION

```
// Initialize Public Key Containers
LPKeyPair<DCRTPoly> keyPair;

// Generate a public/private key pair
keyPair = cryptoContext->KeyGen();

// Generate the relinearization key
cryptoContext->EvalMultKeyGen(keyPair.secretKey);

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

STEP 3 – ENCRYPTION

```
// First plaintext vector is encoded
std::vector<int64_t> vectorOfInts1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
Plaintext plaintext1 = cryptoContext->MakePackedPlaintext(vectorOfInts1);
// Second plaintext vector is encoded
std::vector<int64_t> vectorOfInts2 = {3, 2, 1, 4, 5, 6, 7, 8, 9, 10, 11, 12};
Plaintext plaintext2 = cryptoContext->MakePackedPlaintext(vectorOfInts2);
// Third plaintext vector is encoded
std::vector<int64_t> vectorOfInts3 = {1, 2, 5, 2, 5, 6, 7, 8, 9, 10, 11, 12};
Plaintext plaintext3 = cryptoContext->MakePackedPlaintext(vectorOfInts3);

// The encoded vectors are encrypted
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, plaintext1);
auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey, plaintext2);
auto ciphertext3 = cryptoContext->Encrypt(keyPair.publicKey, plaintext3);
```

STEP 4 – EVALUATION

```
// Homomorphic additions
auto ciphertextAdd12 = cryptoContext->EvalAdd(ciphertext1, ciphertext2);
auto ciphertextAddResult =
    cryptoContext->EvalAdd(ciphertextAdd12, ciphertext3);

// Homomorphic multiplications
auto ciphertextMul12 = cryptoContext->EvalMult(ciphertext1, ciphertext2);
auto ciphertextMultResult =
    cryptoContext->EvalMult(ciphertextMul12, ciphertext3);

// Homomorphic rotations
auto ciphertextRot1 = cryptoContext->EvalAtIndex(ciphertext1, 1);
auto ciphertextRot2 = cryptoContext->EvalAtIndex(ciphertext1, 2);
auto ciphertextRot3 = cryptoContext->EvalAtIndex(ciphertext1, -1);
auto ciphertextRot4 = cryptoContext->EvalAtIndex(ciphertext1, -2);
```

STEP 5 – DECRYPTION

```
// Decrypt the result of additions
Plaintext plaintextAddResult;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextAddResult, &plaintextAddResult);

// Decrypt the result of multiplications
Plaintext plaintextMultResult;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextMultResult, &plaintextMultResult);

// Decrypt the result of rotations
Plaintext plaintextRot1;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextRot1, &plaintextRot1);
Plaintext plaintextRot2;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextRot2, &plaintextRot2);
Plaintext plaintextRot3;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextRot3, &plaintextRot3);
Plaintext plaintextRot4;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextRot4, &plaintextRot4);
```

Explains some non-primitive operations available in PALISADE and further encoding-related 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

MULTIPLE WAYS TO DO ADDITION

- Let us say we need to add 128 integers.
  - Assume the ciphertext dimension is 4K.
  - How we pack the data to get the most efficient result?
- Option 1 (Internal Addition)
  - Pack 128 integers into a single ciphertext and run **EvalSum**.
  - This requires log 128 = 7 rotations (rotations are roughly 100x more expensive than **EvalAdd**).
- Option 2 (External Addition)
  - Put each integer into a separate ciphertext: 128 ciphertexts in total.
  - Addition requires 128 EvalAdds (vs roughly 700 EvalAdds in Option 1).
  - Much faster than Option 1 but requires 128x storage.
- Depending on the tradeoff between runtime and storage, we may choose option 1 or 2 or a hybrid of the two approaches

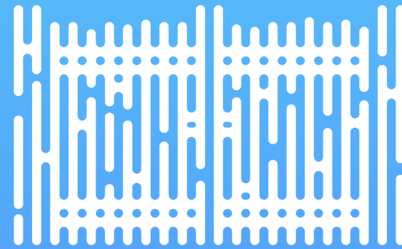
PALISADE

HOW TO ENCODE NON-INTEGERS?

- Any real number can be represented as an integer based on desired fixed precision
- For example, we have a variable in the range [-7,10] and we need to support 2 decimal digits of precision
  - We can encode -7.00 as -700, -6.99 as -699, -6.98 as -698, …, 9.99 as 999, and 10.00 as 100.
  - We need to choose p such that any input/intermediate/output values lie between $-p/2$ and $p/2$.
  - The result needs to be scaled down to compensate for the initial scaling and any multiplications during the computation.
- Limitations of this approach
  - Only exact computations are supported: to support a larger magnitude of integers (more computations), we need to choose a larger plaintext modulus p.
  - Approximate homomorphic encryption is a much better option for this (next month's webinar).

SELECTING CIPHERTEXT MODULUS REVISITED

- Typically the ciphertext modulus is determined by the multiplicative depth
  - However there are applications where we have a large number of additions
  - For example, when we represent a scalar multiplication as many additions to use a smaller ciphertext modulus
- In this case, a large number of additions (thousands) can be equivalent to one or more multiplications
  - We can compute an effective depth to account for the extra noise introduced by many additions
- In such scenarios, BFV is the scheme to use (BGV is built around the multiplicative depth)



THANK YOU

ypolyakov@dualitytech.com