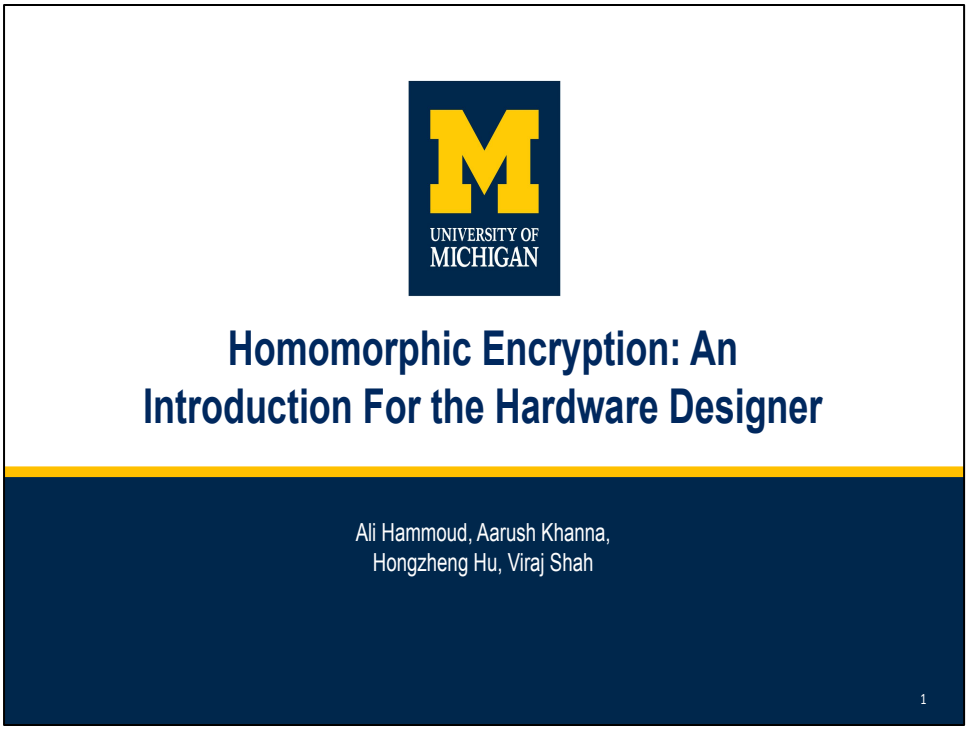


Focus on Brakerski/Fan-Vercauteren (BFV) homomorphic encryption scheme, designed to help electrical engineers (and non-cryptographers) understand FHE.

This presentation is a partial explanation of the algorithm. See the “pymodel” for implementation. We link to implementation of each component

Outline

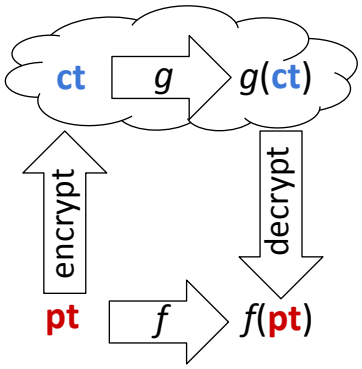
- Introduction & Algorithms Background
- Algorithms Detail (no RNS)
- Algorithms Detail (RNS)
- Hardware Considerations
- References

Outline

- Introduction & Algorithms Background
- Algorithms Detail (no RNS)
- Algorithms Detail (RNS)
- Hardware Considerations
- References

Fully Homomorphic Encryption (FHE)

- Encryption scrambles data ensuring confidentiality
- *But* most algorithms require decryption before data processing
- FHE does compute on encrypted data
  - Plaintext (pt): unprotected data
  - Ciphertext (ct): scrambled data
    - ct = encrypt(pt)
  - **Goal: plain text compute equal to compute on cipher text**
    - e.g. pt_1+pt_2= decrypt(ct_1+ct_2)

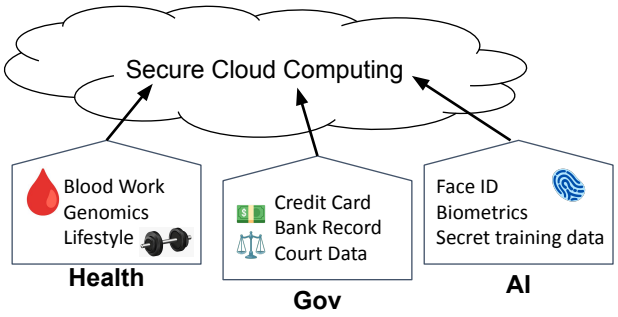


FHE can implement (almost) any function "*f*" without decrypting data

Encryption Algorithms scramble data ensuring confidentiality, but most algorithms require decryption before data processing. This means data processing in the cloud requires trusting the service provider. (if the cloud server must compute on the data) One solution is Homomorphic Encryption: a scheme that preservers computations on encrypted data. Throughout these slides, we refer to unprotected data as "plaintext" or "pt", and encrypted data as "ciphertext" or "ct". To produce a cipher text, you must encrypt the plaintext, and we will explain the encryption process in later slides. The opposite of encryption is decryption, which allows you to produce a plaintext from a ciphertext (if you have the "password" -> secret key).

Motivation & FHE Applications

- Secure cloud computing
  - Confidential machine learning
  - Secret health data on the cloud
  - Database processing of government records
- We demonstrate IoT sensor data preprocessing

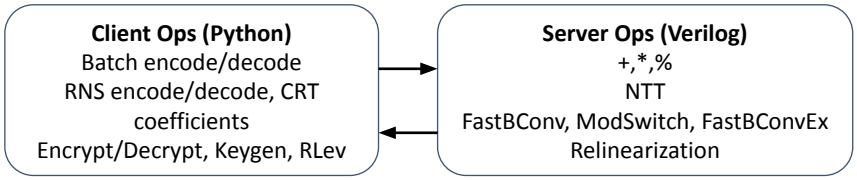


FHE is useful for secure cloud computing. (We're just doing a simple example and testing for correctness) In our repository, as a toy example to prove our implementation, we demonstrate SIMD preprocessing of sensor data from edge devices. For example, we will normalize and apply calibration algorithm for many sensor temp/humidity samples.
Besides the applications shown here, FHE may also be useful in IP protection.

In practice, FHE is far too slow for most applications. Hardware acceleration may be a solution to more practical FHE.

Brakerski/Fan-Vercauteren (BFV) Scheme

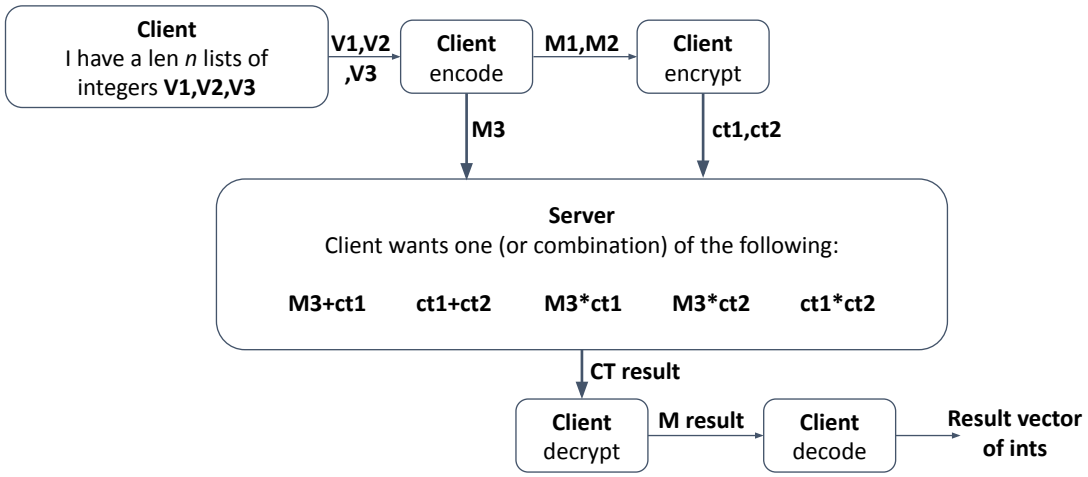
- Introduced in 2012 “second gen” FHE [1]
  - Integer processing with algorithmic **SIMD** (many independent INT ops)
  - 1000x Parallel speedup (e.g. n=1024)
- **We implement addition and multiplication on the server**
- RLWE: Ring Learning With Errors (lattice based cryptography)
- Testing infrastructure (client): verify correctness by comparing plaintext calculations with encrypted calculations



This presentation focuses on the BFV scheme. Introduced in 2012, the scheme includes SIMD processing for parallel speedup. FHE operation latency is very large; SIMD processing built into the algorithm increases throughput efficiently. As we explain later, several values are combined into a single ciphertext with computations on the ciphertext producing several results.
Our implementation includes the BFV scheme encrypt/decrypt and addition/multiplication ops. The RTL implementation includes only addition and multiplication, as a service provider would not require encrypt/decrypt acceleration server-side.

BFV Basic Workflow

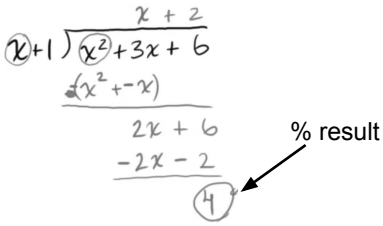
- This presentation covers the following workflow



We will explain how to implement the following operations in BFV. Suppose we have a client who has 3 vectors of integers, each of length n.
1) before encryption, the client will encode these vectors (we will explain the reason behind this on the next slide)
2) Suppose the client will encrypt two of these vectors and keep the third vector as a plaintext
3) the plaintext and both ciphertexts are sent to the server
4) the server can do: ct-ct addition, ct-pt addition, ct-ct multiplication, and ct-pt multiplication (based on whatever algorithm/program the client wants to run on the server)
5) server sends result back to client
6) client decrypts the result
7) client decodes the result to see the output of the program they ran on the server

Mod Polynomials (Background 1)

- BFV produces n parallel INT ops (e.g. n=1024) modulo the **plaintext modulus "t"**
  - e.g. **t=2^16** for INT16 arithmetic
- Encodes plaintext integers in the coefficients of a polynomial
  - **Entire polynomial is modulo x^n+1** (e.g. $x^{1024}+1$)
    - Hint: think back to polynomial division you learned in grade-school



Polynomial Long Division Example
See noRNS/BFV_Config.py:71

Integer modular arithmetic wraps around
9+4 % 12 = 1

At its mathematical core, BFV encodes plaintext data as polynomials. Each integer you encrypt fills a coefficient in a large polynomial. You as the user will see integer arithmetic modulo t which is the plaintext modulus. You may already know about int mod arithmetic, but as a refresher, we mean wrapping arithmetic (you can see the figure below like clock face logic).
We said that BFV encodes polynomials, these polynomials are mod x^n+1 so that successive convolutions don't keep producing longer and longer polynomials. What does this mean? Instead of plain numbers, calculations "wrap around" both in the coefficients like I just said and in the degree (mod x^n+1). It's similar to how you learned polynomial division—we constantly take the remainder of polynomial arithmetic results. Practically, we are not doing poly division after each multiplication, there are more efficient algorithms.

When we mention polynomial multiplication, we typically mean negacyclic convolution. The "naive" approach for negacyclic convolution is to compute the full convolution (length 2N-1), then modulo demonstrated on the left with "Polynomial Long Division" remainder. The negacyclic convolution is the remainder after a polynomial division of 2with modulus (typically x^n+1), as we will explain later

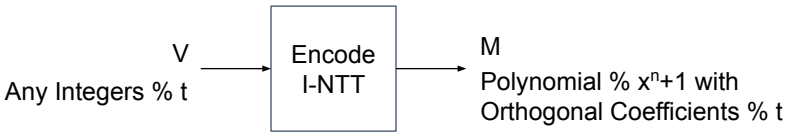
BFV Encoding (Background 2)

Why not directly interpret integer vector "V" as polynomial coefficients "M"?

- SIMD computation, so we do NOT want coefficients to interfere with each other
  - Coeffs would 'bleed-into' each other during cyclic convolutions (poly mult)
- **Solution: encode "V" in an orthogonal basis**

What is a convenient orthogonal basis we can use?

- ❌ DFT complex roots of unity
- ✅Number Theoretic Transform (NTT) primitive roots of unity (mod arithmetic)
  - Root of unity: pick w such that $w^n = 1\%q$ AND $w^k \neq 1\%q$ for $k<n$ (we actually use 2n-th roots)



See noRNS/BFV_Config.py:87, generic_math.py:25

Like I said before, we want many independent SIMD operations, so if we directly take our desired integers as polynomial coefficients and try to run BFV scheme, it will produce wrong results. We will need to do cyclic convolutions so if we don't use a special encoding the coefficients will bleed into each other.
What we need is an orthogonal basis. Obviously we are not dealing with complex numbers and DFT is not appropriate, but there is a similar transformation called Number Theoretic Transform (NTT)—the integer modular version of the DFT. In summary, we run NTT transformation (actually the inverse transformation) to ensure independent SIMD before we can start doing our BFV operations.

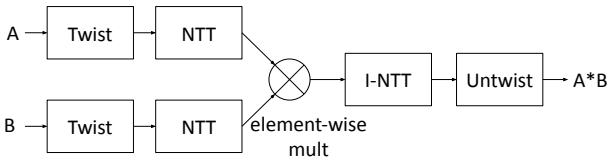
Negacyclic Convolution (Background 3)

- Used for polynomial multiplication mod x^n+1
- Naive implementation: polynomial expansion & negative wrapping (noRNS/BFV_Config.py:71)

Example:

$$(x^2+x+1)*(2x^2+3x+4) \% x^3+1$$
$$(2x^4+3x^3+4x^2+2x^3+3x^2+4x+2x^2+3x+4) \% x^3+1 =$$
$$(2x^4+5x^3+9x^2+7x+4) \% x^3+1 =$$
$$9x^2+(7-2)x+(4-5)$$
$$9x^2+5x-1$$

- More efficient INTT{NTT(a)*NTT(b)}
  - Twist discussed later



When we mention convolution or polynomial multiplication, we are usually referring to negacyclic (or negative wrapping) convolutions. The naive, slow way of doing this is to calculate a full 2n-1 length convolution then shrink it back with negative wrapping. Here is an example for clarity.

There is a more efficient method making use of a transformation called NTT which, much like DFT, has the nice property where convolutions are equivalent to element wise multiplication in the transform domain. To implement efficient multiplication, start with NTT to transform each polynomial, multiply element wise, then invert the transform. The “twist” and “untwist” steps are similar to the FFT twist factors used for inverting the transform.

Outline

- Introduction & Algorithms Background
- Algorithms Detail (no RNS)
- Algorithms Detail (RNS)
- Hardware Considerations
- References

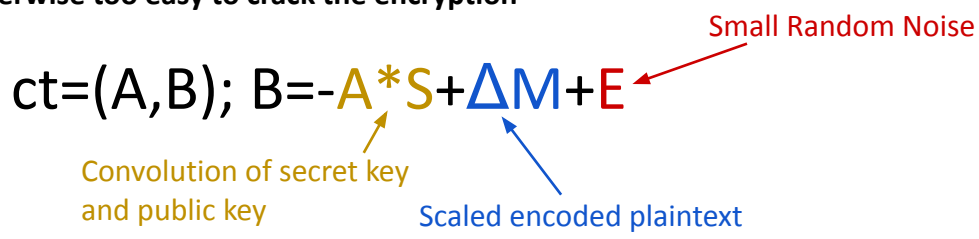
BFV Parameters

M: encoded plaintext message polynomial. Encoding is done by $M=NTT^{-1}\{\text{my vector of ints}\}$
n: degree of polynomials used + 1 (n coefficients in n-1 degree polynomials)
t: a prime number or power of prime number (p^r). See how to pick here
q: cipher text modulus, q much larger than t and t divides q
Δ aka Delta=floor(q/t): the scaling factor of the plaintext
S: randomly generated secret key is an n-1 degree polynomial where the coefficients are randomly chosen ternary {-1,0,1} or binary {0,1}. In total there are n coefficients in the secret key
RLev(S^2): is a set of relinearization keys calculated from S2, explained later
RLev(S): is a set of bootstrapping keys calculated from S
A: one time randomly generated public key generated for each encryption. A is an n-1 degree polynomial where the coefficients are modulo q
E: the noise polynomial is an n-1 degree polynomial whose coefficients are small numbers modulo q and randomly chosen from a gaussian distribution

We will use these parameters throughout the presentation. We will explain each in more detail / where they are used. Come back to this slide as a reference.

BFV Encryption

- Before encryption raise polynomial "M" to modulo **"q" (cipher text modulus)**
  - q is typically **300-600 bits** for good security
  - Implement by multiplying each coefficient by Δ**=q/t**
- **Keygen**: define random secret key "S" (you reuse many times) and random public key "A" generated for every encryption. See BFV_model.py:16,43
- **Encryption adds noise** zero-centered small variance gaussian "E" modulo q
  - **Otherwise too easy to crack the encryption**



Encryption is the process of scrambling data. First, we multiply every coefficient in the polynomial by a scaling factor delta to scale from plaintext modulus t up to a very large cipher modulus "q". q is typically 300+ bits for good security.
For key generation, we generate a random secret key "S" (keep for many encryptions), and each time we want to encrypt a vector of integers we need a new random public key "A" that we will share with everyone.
The encryption process multiplies "S" with "A" (negacyclic convolution) then adds the scaled plaintext polynomial deltaM. Crucially, we add a little bit of random noise "E". This noise is needed to make the scheme more secure so that attackers can't easily break the encryption. The final ciphertext is the pair of polynomials A comma B (where B is the new public key we calculated).

BFV Decryption

- Reverse the encryption process (calculate M from the ct using secret key)
  - Noise must be small otherwise decryption does not work
- Once we have “M” we decode (forward NTT) to recover our integers.

ct=(A,B), S=Secret Key

$[(B+A*S \bmod q)/\Delta] \bmod t = [(\Delta M+E)/\Delta] \bmod t$

As long as E is small, we can round…

$[(B+A*S \bmod q)/\Delta] \bmod t \sim M \bmod t$

See BFV_Model.py:65

BFV Bootstrapping

- Leveled HE vs Full HE: *Seems like noise will limit our computational depth…*
  - **Solution: reset the noise via "bootstrapping"**
    - Client gives server RLev(S) evaluation keys, encryption of "S"
- Leveled HE does not implement bootstrapping
  - Instead just be careful about not exceeding maximum computation depth
  - If you need more depth, increase Δ
  - **BFV supports full, but we implement leveled** (no bootstrapping)

You might have guessed from the last slide that adding noise will make it harder to decrypt the data. That does happen so when we do operations, encrypted data accumulates noise. If noise grows too much, the ciphertext becomes impossible to decipher. The solution to support arbitrary depth is an operation called "bootstrapping."
What we are actually doing does not implement bootstrapping, instead we are doing a "leveled" scheme. With a leveled scheme, you have to monitor computation depth and set parameters appropriately before calculations to ensure noise doesn't grow out of control.

CT-CT Add & CT-PT Add

CT-CT Add, see BFV_model.py:87

- ciphertext(Message1)=(A1,B1) and ciphertext(Message2)=(A2,B2)
- **ciphertext(Message1+Message2) = (A1+A2, B1+B2) mod q**

CT-PT Add, see BFV_model.py:92

- Refresher: ct encrypts a message M which encodes n integers. n is the degree of the polynomial used in BFV. We can add n plaintext integer constants to our encrypted message M.
- Form a new n degree plaintext polynomial (call it Λ) which is the encoding of our plaintext vector of integers we want to add
- If ciphertext(M)=(A,B), then **ciphertext(M+Λ) = (A, B+$\Delta\Lambda$) mod q**

CT-PT Multiply

See BFV_Model.py:99

- We can multiply n plaintext integer constants to our encrypted message M.
- Form a new n degree plaintext polynomial (call it Λ) which is the encoding of our plaintext vector of integers we want to multiply
- **ciphertext(M*Λ) = (A*Λ, B*Λ)**

CT-CT Multiply Overview

- More difficult than the other operations
- summary (next slide) then explanations of each step in more detail
- RNS version of CT-CT Multiply is more intuitive (if you want to skip there)

CT-CT Multiply Dataflow

See noRNS/BFV_Model.py:139

1. Start with ct1=(A1,B1), ct2=(A2,B2) modulus "q" (the ct modulus)
2. **Binomial Mult**: D0=B1*B2, D1=B2*A1+B1*A2, D2=A1*A2
   a. Do NOT apply modulo ct modulus "q". **Use a larger modulus "Q" (q*Delta)**
3. **Relinearization**: compute ciphertexts (ct)
   a. ct_{α} = D1 , D0
   b. ct_{β} = < $Decomp^{\beta,\ell}(D2)$, $RLev_{S,\sigma}^{\beta,\ell}(S^2)$ >, where <> is the inner product
      i. Decomp and RLev are both lists of cipher texts, do dot product between cipher texts by treating the A and B portions as completely separate
   c. $ct_{\alpha+\beta}$= ct_{α} + ct_{β}
4. **Rescaling**: update $ct_{\alpha+\beta}$ to **ct'** = ($A_{\alpha+\beta}/\Delta$, $B_{\alpha+\beta}/\Delta$) mod q

The result is $ct_{\alpha+\beta}$

RLev

- When the client creates the secret key S, they also publish a set of encryptions of S^2 multiplied by every gadget vector element
  - This set of encryption keys are called relinearization keys
- RLev(S^2)=encryption of $\underline{B}*s^2$
  - This produces a cipher text (A_{rlev},B_{rlev})
- $Decomp_{\beta,l}(D2)$ = some cipher text (A_{decomp}, B_{decomp})
- ct_{β}= (A_{decomp}*A_{rlev} , B_{rlev}*B_{decomp})

Number Decomposition Intuition

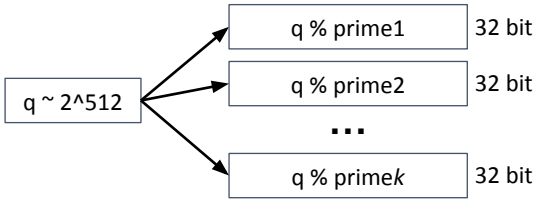
- Number decomposition is a special case of gadget decomposition
- Suppose I have some integer "γ".
  - To do gadget decomposition(γ) in base 10 (decimal), we get…
    - γ=g0 + 10*g1 + 100*g2 + 1000*g3 + …
      - capture every position in γ
  - I can do gadget decomposition in any base "w", typically we pick a base that is a power of 2
  - I recover γ using dot product with the **g** vector γ=<**g**,Decomp(γ)>
- Suppose I have polynomial P, I can do number decomposition on every single coefficient in P. Notate this as Decomp(P)
  - This is sometimes written as $\text{Decomp}^{\beta,\ell}(P)$. The result of this is a matrix
    - β represents the decomposition base (e.g. decimal is base 10, binary is base 2)
    - ℓ (lower case L) represents the number of decomposition levels
      - E.g. in decimal, if I want to represent numbers up to 10000 I need ℓ=5 levels

Outline

- Introduction & Algorithms Background
- Algorithms Detail (no RNS)
- Algorithms Detail (RNS)
- Hardware Considerations
- References

Residue Number System (RNS)

- Mentioned previously q~300-600 bits
  - ❌ Don't want 700 bit multipliers in our pipeline
- Solution: ✅RNS for parallel representation of large integers
  - Fully demonstrated on BFV in 2016 by Bajard,Eynard,Hasan,Zucca (BEHZ) [2]
- Idea: big modular arithmetic as combination of many small prime mod arithmetic
  - Chinese Remainder Theorem (CRT) guarantees every int in the range has a unique representation modulo the primes. See sympy.ntheory.modular.crt



Breakup a very large q (e.g. 512 bits) into smaller 32 bit datapaths

As I said before, the cipher text modulus “q” is hundreds of bits. Obviously we don't want 300 bit multipliers in our pipeline (you need even more when you start doing multiplication), so thankfully in 2016 some researchers figured out how to apply the residue number system RNS scheme to BFV. RNS represents big numbers as a combination of small, parallel chunks using different prime moduli.
Thanks to the Chinese Remainder Theorem, it is perfectly correct to do all arithmetic completely in parallel without any communication, and reconstruct the final result at the end. Here in this figure, I show that we can split big INT compute into many parallel 32 bit data paths each with different prime modulo.

RNS Arithmetic

- Most operations stay the same & implemented completely in parallel
  - 512 bit addition becomes 16x parallel 32 bit add
  - 512 bit mult becomes 16x parallel 32 bit mult
- Sometimes we need to mod switch (e.g. for mult 512 bit is not big enough)
  - **Mod raise**: add extra 32 bit primes to represent a larger number
  - **Mod switch**: remove 32 bit primes to go to a smaller representation
  - **All Ops possible without large integers**
    - Practically both operations can reuse same hardware operating in different modes

Visualizing a Polynomial With RNS Coefficients

- RNS makes each coefficient several 32bit INTs long.
  - Simply replace each coefficient with its RNS representation
- In example below, suppose I have my RNS representation has k residues.

My_polynomial_A = {coef1, coef2, coef3, … , coef*n*}

residue#1	residue#1	residue#1	residue#1	% Basis Prime Integer #1
residue#2	residue#2	residue#2	residue#2	% Basis Prime Integer #2
residue#3	residue#3	residue#3	residue#3	% Basis Prime Integer #3
…	…	…	…	
residue#*k*	residue#*k*	residue#*k*	residue#*k*	% Basis Prime Integer #*k*

RNS Basis

- Will use 3 different RNS basis. See BFV_config.py:24
  - Each basis is composed of prime numbers ~ 32 bits large
- q-basis: represents integers modulo the ciphertext modulus “q”~ 300-600 bits
  - q-basis requires consists 10-18 * 32 bit primes
- B-basis: similar in size to the q-basis. q*B is large enough to hold product of two integers which are modulo q.
  - q*B is 600-1200 bits. Requires 20-36 * 32 bit primes
- Ba-basis: typically implemented with a single 32 bit prime
  - Used for fastBConvEx. Explained later (dropped during the algorithm)

Combinations of these bases e.g. qBBa, BBa, etc.. are also used

CT-CT & CT-PT Add, CT-PT Mul

- All algorithms stay the same. No modifications from non-RNS versions
  - All operations performed in the “q” basis
- You will need to implement the following operations
  - RNS_INT * RNS_INT
  - RNS_INT + RNS_INT
  - RNS_INT * small constant int

CT-CT Mul

See BFV_Model.py:165. Start with ct1=(A1,B1), ct2=(A2,B2) with RNS coefficients in the "q" basis

1. **Mod Raise**: raise the modulus of all coefficients of A1,B1,A2,and B2 from q-basis to qBBa basis
2. **Binomial multiplication**: D0=B1*B2, D1=B2*A1+B1*A2, D2=A1*A2
3. **Constant Multiplication by t**: multiply all coefficients of D0,D1,and D2 by "t" (the plaintext modulus)
4. **Mod Switch**: switch the modulus of all coefficients of D0,D1,and D2 from qBBa basis to BBa basis
5. **FastBConvEx**: from BBa back to the q-basis
6. **Relinearization**: do the relinearization step in the q-basis

RNS Base Conversions (Intro)

- Must change the modulus for ct-ct multiplication.
  - Normal int mod change is easy, but how to do hardware efficient RNS?
- *Maybe design an RNS constructor block?* Build int using CRT Reconstruction
  - ✅ Precompute y_i and z_i (slow, but replace with Look Up Table)
  - ❌ intermediate sum is huge

```
PartialProducts = list
for i in len(RNS_basis):
    yᵢ  =  q / qᵢ
    zᵢ  =  yᵢ⁻¹  (mod qᵢ)
    PartialProducts.append(xᵢ · zᵢ · yᵢ)
BigInteger = Sum(PartialProducts) % q
RNSReconstruct(BigInteger, newBasis)
```

Heavy compute replaced with LUT

Big number!

What we want

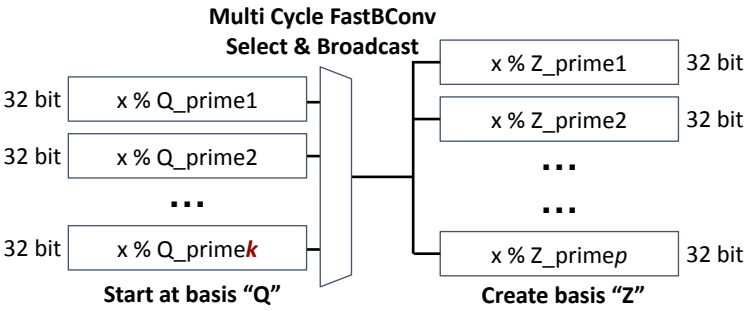
Just like with “normal” integer multiplication, the result of multiplication of individual coefficients will be a much larger integer. Thus, we must extend the RNS integers so that they are large enough to contain a product result of two ciphertext coefficients. In the RNS representation, a larger integer can be represented by adding more residues / expanding the RNS basis.

- The normal int mod change is simple, but RNS mod change is more involved
- Naive implementation: What a possible implementation? - RNS constructor using CRT Reconstructional algo
  - Involves compute heavy steps - replace with LUT
  - Drawback is that it requires big integers
- There is a better way

Fast Base Conversion

- FastBConv: Better base conversion hardware
  - Directly convert between RNS primes **(no big sums)**
  - **Tradeoff: little accuracy lost**
- Modified CRT compute $x_i \cdot z_i \cdot y_i$ % Z_prime***j***
  - Never let sum get large
  - Bounded rounding error
- Broadcast coef. & LUT $z_i \cdot y_i$
  - 1 calculation every **k** cycles
    - Accumulate result

See generic_math.py:262



FastBConv: direct conversion, some rounding

The fastBConv algorithm allows for inexact but highly efficient base conversion without converting back to absolute (very large 300+ bit) integers. The algorithm is inexact, but as discussed in a future slide, there is an additional modification to allow for exact conversions when needed (at a cost of sacrificing one of the residues).

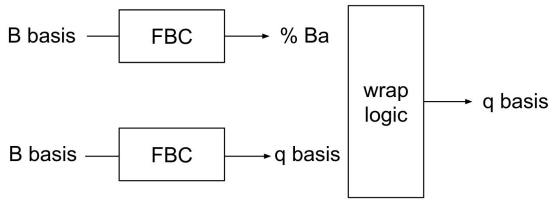
Directly convert between RNS primes with no big intermediate sums - tradeoff: lose a little accuracy
Modified CRT computation
applies reconstruction on partial sums to avoid getting a big intermediate sums
Still use LUT for zi and yi

Mod Drop, Mod Switch, Mod Raise

- Shrink-the-Modulus Procedures:
  - ModDrop: drop primes (no calculations needed)
    - Only works if q’ divides q
  - ModSwitch: Reduce RNS modulus from q->q’ and scale value by q’/q.
    - see generic_math.py:282, uses ModDrop and one fastBConv
    - An efficient way to divide the modulus AND the integer simultaneously
- Increase mod procedure: ModRaise
  - Is actually done using a single fastBConv (see BFV_model.py:170)
  - Switch to a larger basis, with some additional noise from FastBConv
  - q -> q b ba

Fast Base Conversion Exact

- FastBConv: Better base conversion hardware
  - Directly convert between RNS primes **(no big sums)**
- BBa->q
- See [generic_math.py:311]



This block is required in the last step of ct-ct multiplication to ensure that the final result has an acceptably low conversion noise. Exact fast base conversion can be achieved by sacrificing an RNS prime modulus. The prime that is dropped is called the “Ba” modulus. We implement this using two parallel fast base conversion blocks and wrap logic as shown in the simplified diagram below.

Relinearization

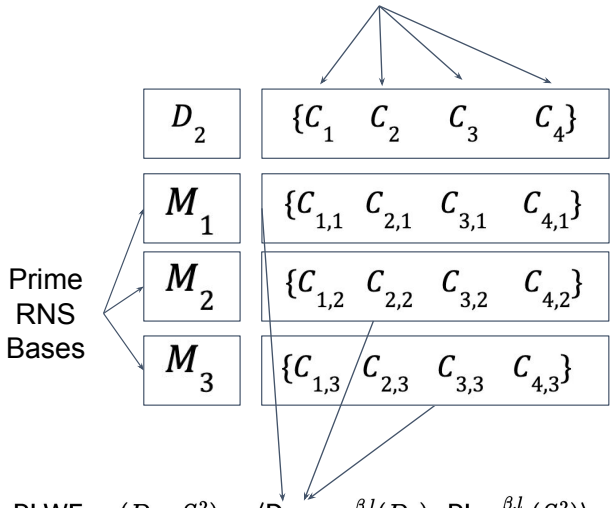
- Decomposition Mult (RNS)
  - **Input**: CT1, CT2 -> D0, D1, D2 **(NTT)**
  - To Avoid Noise -> Gadget Decomp
    - No Computation Necessary - **Only Wires**
  - Inner Product: Element Wise - Polynomial Mul
    - Fast with **pipelined NTT**
    - Stream outputs & mult in **parallel**
- Relin (a.k.a RLev) Keys (RLWE)
  - Array of ciphertexts - Provided by Client
  - **Input** to Relinearization Block

$$\underbrace{B^{\langle 1\rangle}B^{\langle 2\rangle}}_{D_0} + \underbrace{(B^{\langle 2\rangle}A^{\langle 1\rangle} + B^{\langle 1\rangle}A^{\langle 2\rangle})}_{D_1} \cdot S + \underbrace{(A^{\langle 1\rangle} \cdot A^{\langle 2\rangle})}_{D_2} \cdot S \cdot S$$

$$ct_{\alpha} = (D_0, D_2) \qquad ct_{\beta} = RLWE_{S,\sigma}(D_2 \cdot S^2)$$

Output $ct_{out} = ct_{\alpha} + ct_{\beta}$

RNS Residues (Already Computed)

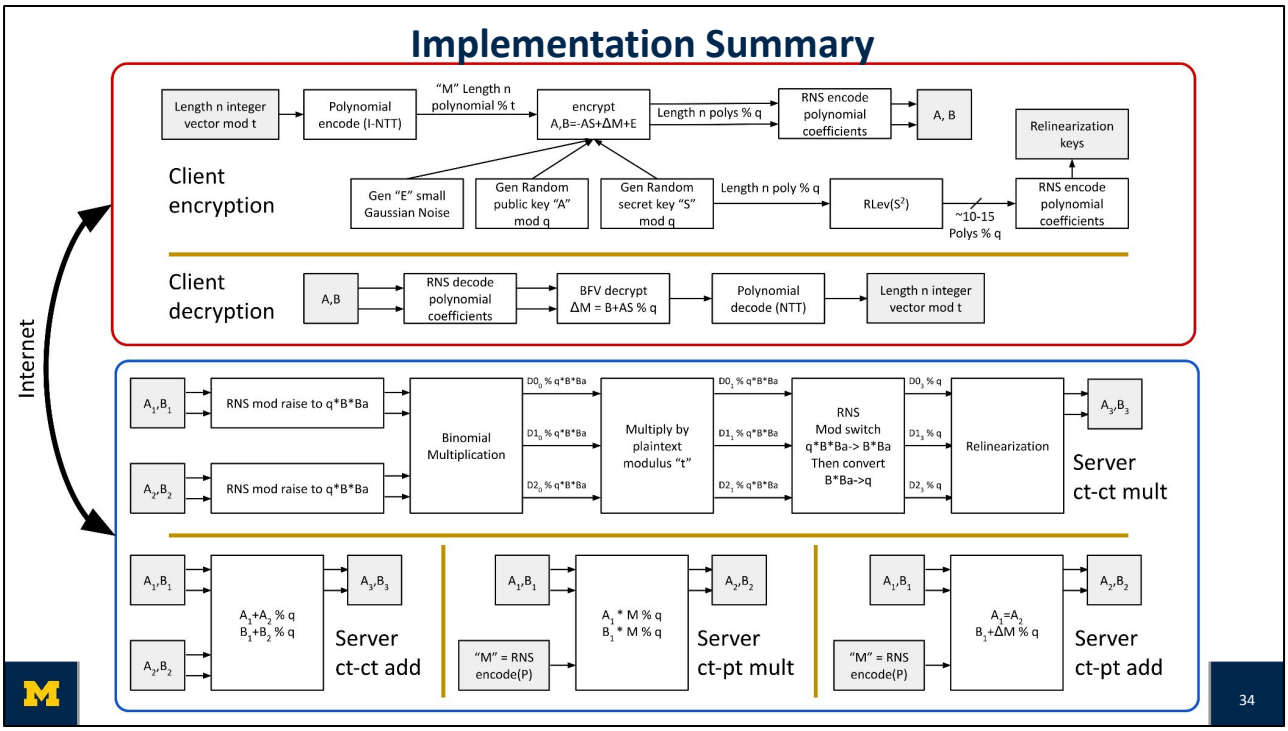


$$\text{RLWE}_{S',\sigma}(D_2 \cdot S^2) = \langle \text{Decomp}^{\beta,l}(D_2),\ \text{RLev}^{\beta,l}_{S,\sigma}(S^2)\rangle$$

Inner Product

CT-CT steps:

- Mod Raise q -> qbba
- Polynomial Multiplication (D0, D1, D2)
- Constant Multiplication (plaintext modulus t)
- Mod Switch qbba -> bba
- FastBConvEx bba -> q
- Relinearization



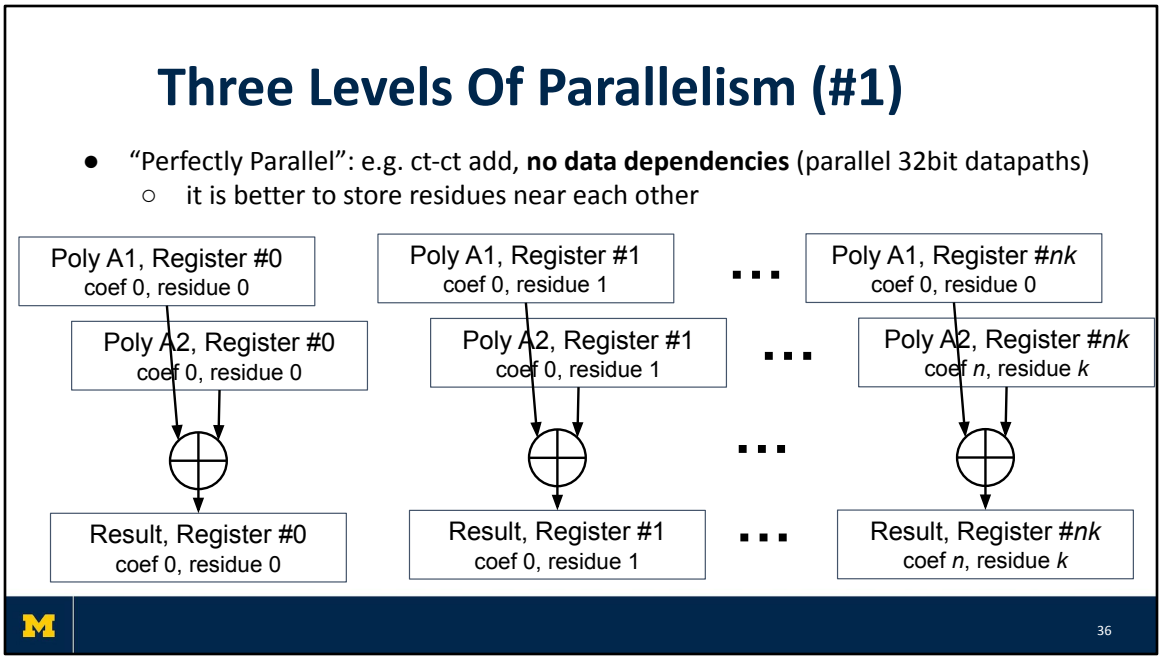
As shown previously, encryption produces two polynomials, A and B, along with some other encrypted data we need to send to the server (like the relinearization keys which we will explain later).

- First, If we want to add 2 cipher texts we can directly add their respective A,B polynomials coefficient wise. With RNS each coefficient operation is broken into smaller parallel residue operations.
- Next, If we want to add a cipher text with plain text we can do so by adding the plaintext to the B polynomial.
- Then, to multiply cipher text and plain text we multiply both A and B by the plain text. Polynomial multiplication is the same as the convolution we explained before, so we use NTT to accelerate this step.
- After each operation we always do coefficient wise / parallel mod q to keep all elements mod q
- Lastly, the most complicated operation we implement in RTL is the ct-ct mult. You can see this has several steps.
    - Firstly, we need to do a mod raise because multiplying two cipher text we will need a larger store. You can see we have to raise from modulo q arithmetic to q*B*Ba. So the q we already know is the cipher text modulus. The B here represent the additional scaling we need to represent a ct-ct product, lastly the Ba we have not yet introduced. We will talk more about Ba when we talk about fast base conversion for RNS. So actually if the ciphertext is 500bits per coefficient, then the q*B*Ba represnetaiotnn is like 1500bits per coefficient. We are representing each of these moduli as a single variable, but in reality

- because we are using RNS, each moduli is a set of prime numbers, and when we want to add or remove a set of primes we are doing operations to rewrite the big integer under the different RNS representations.
- After increasing to a larger representation. We do binomial multiplication like (A1+B1)*(A2+B2). This gives us 3 terms. We get D0 which is B1*B2, D1 which is the cross terms (A1*B2+A2*B1), and lastly we get D2 which is B1*B2. As we will discuss later, D2 needs extra processing steps when we want to combine these 3 terms back into a single cipher text
- After binomial multiplication, we want to get back to a single cipher text and we also want to scale back to modulo q. The next two steps bring us back to modulo q, and finally the relinearization step gives us a single cipher text (A3,B3)

Outline

- Introduction & Algorithms Background
- Algorithms Detail (no RNS)
- Algorithms Detail (RNS)
- Hardware Considerations
- References

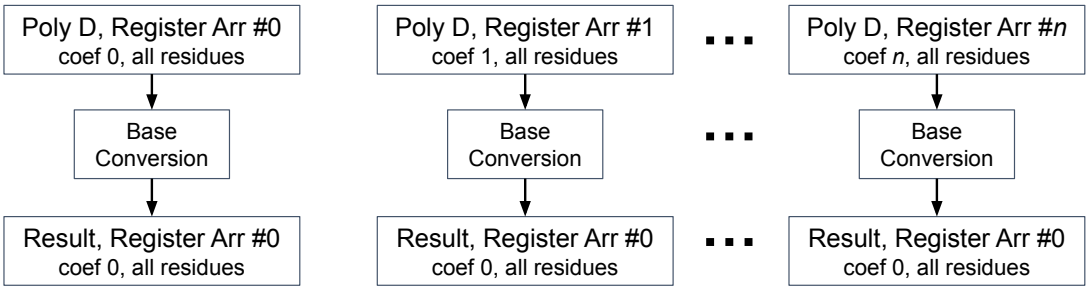


One type of parallelism in the BFV algorithm:

- Perfectly parallel: this is completely parallel. For example, in ct-ct addition, you can add each residue of each coefficient in both polynomials residue-wise with NO data dependency between residues
  - Complete parallelism is found in several operations, and would seem to encourage storing residues in hardware near each other

Three Levels Of Parallelism (#2)

- "Perfectly Parallel": e.g. ct-ct add, **no data dependencies** (parallel 32bit datapaths)
- "Coefficient wise": **inter-residue dependencies**, coefficient level parallelism

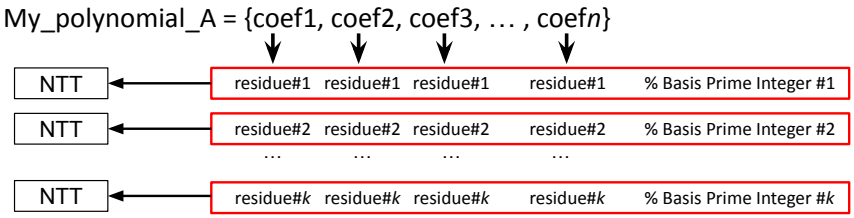


One type of parallelism in the BFV algorithm:

- "Coefficient wise": there is a data dependency between residues, but coefficients are completely independent. this is seen in the base conversions: fastBConv, mod switch, fastBConvEx

Three Levels Of Parallelism (#3)

- "Perfectly Parallel": e.g. ct-ct add, **no data dependencies** (parallel 32bit datapaths)
- "Coefficient wise": **inter-residue dependencies**, coefficient parallelism (base convs)
- "Residue wise": **inter-coefficient dependencies**, residue level parallelism (e.g. NTT)
- Maybe possible but difficult to achieve optimal layout for residue wise & coef wise
  - One solution: do optimal layout for NTT, then use residue bus for each coef. Base conversions are multi-cycle.



One type of parallelism in the BFV algorithm:

- "Residue wise": you can "slice" the n-slot polynomials into single residue slices (n * 32 bit coefficients). You have dependencies between coefficients, but not between different residues for the same slot.

Number Theoretic Transform(NTT) Algorithm

	FFT	NTT
Domain	Complex domain with floating point	Modular domain with integer
Twiddle Factor	$W_N = e^{-j\frac{2\pi}{N}}$ which making $W_N^N = 1$	$W_N = g^{\frac{P-1}{N}} (\mathrm{mod}\ P)$ making $W_N^N = 1 (\mathrm{mod}\ P)$
Equation	$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}$	$X[k] \equiv \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk} \pmod{P}$

For negacyclic conv, twist/untwist is needed, which is only an element-wise mult.

Twist factor: $\psi^2 \equiv W \pmod{p}$

Parameters are pre-calculated and stored in ROM:

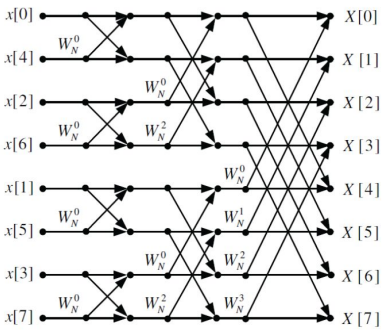
- Prime modular **P** (different from the BFV modular q)
  P=k*N+1 to ensure the primitive root of unity exists.
- Primitive root **g** should follow:
  $g^1, g^2, \ldots, g^{p-1} \pmod{p}$ generate all distinct number from 1 to P-1

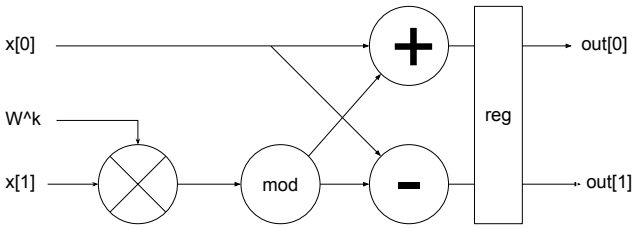
NTT is really similar to the FFT. The only difference is that it is implemented in a field of modulus with integer instead of complex domain with floating point number. Thus no precision loss.

The parameter selection is more complicated for NTT. The primitive root should make the twiddle factor different from each other to make the transformation orthogonal.

HW Implementation of NTT

- Same architecture as the FFT
- *n*-points radix-2 butterfly network
- **Tradeoff**: radix-2 butterfly, more stages shorter critical path. Pipeline between each computing element reduce critical path, increases leakage power / area.
- Critical path: 32bit mod+mul+add



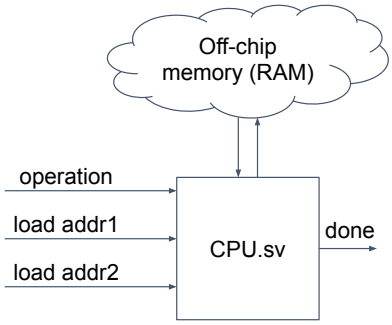


Quite similar with the FFT butterfly module. The only difference is we will need a mod op

The architecture for NTT block is the same as FFT butterfly.

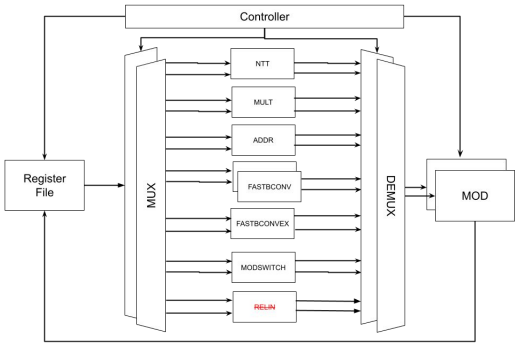
We'll need to choose prime number from Solinas prime(2^k-c), which can change the mod procedure into shift and add. This prime naturally satisfies the NTT requirement of (N*k-1)

Our Architecture: User Perspective



- Simple, proves correctness
- Register file is opaque to the user (managed by controller, like a scratch pad)
- Insert operation, wait till done
  a. Parameters are hard coded (not possible in reality)
  b. No memory in TB (tb writes into the reg file)
- Correct on ct-ct add, ct-pt add, ct-pt mul
- Can be used as “starter-code” for future implementations (open license)

1-Stage Design (no cache)



ct-ct mul -> **~several hundred cycles**

- **Objective: correctness (not efficiency)**
- Microcode style ops (hardcoded into the controller)
- One Functional Unit (or scheduled set) per flow
- Results are routed back to the Register File (one-stage design)

Future Suggestions

- We believe one-stage design is the optimal implementation. Use extra area to implement larger SIMD (algorithmic level efficiency benefits) with much longer crit path, not more regs.
- Future implementations should cut down number of FUs, reuse blocks, consolidate registers -> longer crit path but higher BW + lower power
- Little benefit from modern CPU design (out of order style cores), unless OPs take less cycles

TODOs

1. Implement memory: on board RLev cache and other LUT caches, and user visible on chip memory (so you don't need load/store each op)
2. further reduce the number of functional units. Implement more in microcode
3. Increase crit path to remove regs: switch to radix-4 or radix-8 NTT & make data flow modifications
4. Implement Boot Strapping

We believe one-stage design is the optimal implementation because there is a huge memory demand. Pipelining is extremely wasteful of area/power for little benefit (better to have longer critical path because operation latency is large anyways). Furthermore, it is more optimal to use additional on chip area/memory to implement a larger scheme (larger “n”, more SIMD slots) as that has algorithmic level efficiency benefit. Do not waste area/power on more registers.

As such, we believe an optimal implementation would further reduce the number of functional units (it is possible to combine several), likely switch to radix-4 or radix-8 NTT, eliminate wasteful/redundant logic in the FUs, and address the memory problems (e.g. on board RLev cache and other LUT caches, and user visible on chip memory).

Outline

- Introduction & Algorithms Background
- Algorithms Detail (no RNS)
- Algorithms Detail (RNS)
- Hardware Considerations
- References

References

[1] Fan, J., & Vercauteren, F. (2012). Somewhat Practical Fully Homomorphic Encryption. IACR Cryptol. ePrint Arch., 2012, 144.

[2] Bajard, JC., Eynard, J., Hasan, M.A., Zucca, V. (2017). A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In: Avanzi, R., Heys, H. (eds) Selected Areas in Cryptography – SAC 2016. SAC 2016. Lecture Notes in Computer Science, vol 10532. Springer, Cham.

[3] Ronny Ko. "The Beginner's Textbook for Fully Homomorphic Encryption." arXiv:2503.05136, 2025.

[4] Ardianto Satriawan, Rella Mareta, and Hanho Lee. "A Complete Beginner Guide to the Number Theoretic Transform (NTT)." Cryptology ePrint Archive, Paper 2024/585, 2024.

[5] Jonas Bertels, Michiel Van Beirendonck, Furkan Turan, and Ingrid Verbauwhede. "Hardware Acceleration of FHEW." In 2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pp. 57-60, 2023.

[6] Leo de Castro, Rashmi Agrawal, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. "Does Fully Homomorphic Encryption Need Compute Acceleration?" Cryptology ePrint Archive, Paper 2021/1636, 2021.