Introduction to Homomorphic Encryption

Private AI Bootcamp, Dec 2nd, 2019

Wei Dai

What Is HE? (At U.S. Border Checkpoints)

- A new type of encryption (requires a secret key to decrypt).
- Compute on encrypted data without decryption.
- No one sees results without a secret key.

For chatty officers:

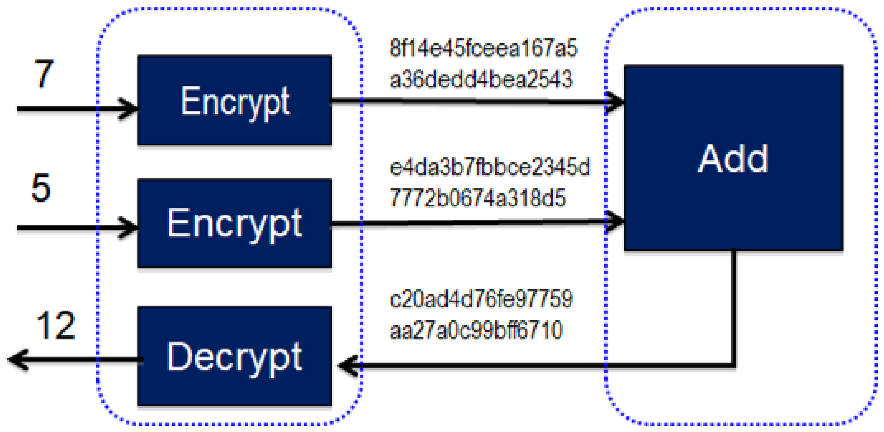
- Since 2009, addition and multiplication are both supported.
- $10^9 \times$ speedup since 2009.
- Still $10^3 \sim 10^6 \times$ slower than unencrypted, but you can …
- Find your birth parents while keeping DNAs secret.

FAQ

- Data enter / stay in / leave untrusted networks encrypted.
- Do operations on ciphertext and plaintext reveal secret?

  No, an operation on ciphertext and plaintext outputs ciphertext.
- Is decryption performed during computation?

  No, computation is performed without decryption.



Popular Schemes

- TFHE: logic gates on bits
- BGV, BFV: exact arithmetic on vectors of numbers
- CKKS: approximate arithmetic on vectors of numbers

Five Stages in HE

- Setup
  - Scheme
  - Security parameters
  - Functionality parameters
- Key generation
  - Secret key, public key, relinearization key, Galois keys
- Encryption
  - A number or a vector of numbers → A ciphertext (2 polynomials)
- Evaluation
- Decryption

SIMD Computation

Message(s) A vector of numbers $\xrightarrow{\text{Batching/Encoding}}$ Plaintext A polynomial $\xrightarrow{\text{Encryption}}$ Ciphertext Two polynomials

7	$7x^0$	$x^{N-1} - 12374x^{N-2} + \cdots$
$(1, 2, 3, 4, \dots)$	$x^{N-1} + 7x^{N-2} + \cdots - 6x^1 + 3$	$x^{N-1} + 57387x^{N-2} + \cdots$

- Enc(1,2,3,4, …) + Enc(1,2,3,4, …) → Enc(2,4,6,8, …)
- Enc(1,2,3,4, …) x Enc(1,2,3,4, …) → Enc(1,4,9,16, …)

- Number of message slots: N in BFV, N/2 in CKKS (in later sessions).
- Batching brings $10^3 \times \sim 10^4$ ×speedup (amortized) and better be used.

Polynomials

- $R = \mathbb{Z}[X]/(X^n + 1)$: $X^n \equiv -1$
- $R_Q = R/Q$: coefficients are computed modulo Q
  - $Q = 5$, coefficients are chosen from $\{-2, -1, 0, 1, 2\}$
- For examples, $n = 3, Q = 5$:
  - $(x^2 + x^1 + 2)(x^2 - x^1 - 2) = x^4 + 0x^3 - 1x^2 - 4x^1 - 4 = -x^2 + 1$
  - $(x^2 + x^1 + 2) + (x^2 - x^1 - 2) = 2x^2$

The most important parameters:
n and $\lceil \log Q \rceil$

Encode and Encrypt

- *Message* is from $\mathbb{Z}_t^n$ in BFV (t: plaintext modulus), from $\mathbb{C}^{n/2}$ in CKKS.
- `Encoder` maps message to a *plaintext* in R_t in BFV, in R in CKKS.
- `Encryptor` maps a *plaintext* to a *ciphertext* in R_Q^2: (ct_0, ct_1)
    - Adds noise to mask secret
    - Random ciphertext each encryption
- Addition/multiplication are preserved in all three forms.

Homomorphic Addition (Hom.Add)

- $(c_0, c_1) = (a_0 + b_0, a_1 + b_1)$ mod q.
- If b is a plaintext, $(c_0, c_1) = (a_0 + b, a_1)$ mod q.
- Plaintexts are added mod t (in BFV).
- $\mathcal{O}(n)$ in terms of integer add mod Q.
- Noise's variance grows $\mathcal{O}(n^{0.5})$ in terms of number of Hom.Add.
- Noise budget is shrunken by 1 bit.

Homomorphic Multiplication (Hom.Mul)

- $(c_0, c_1, c_2) \leftarrow (a_0 b_0, a_0 b_1 + a_1 b_0, a_1 b_1)$ mod q.
- If b is a plaintext, $(c_0, c_1) = (a_0 b, a_1 b)$ mod q.
- Plaintexts are multiplied mod t (in BFV).
- Decrypt requires: $[ct_0 + ct_1 s + ct_2 s^2]_q$.
- $\mathcal{O}(n \log n)$ in terms of integer add / mul mod Q.
- Noise's variance grows $\mathcal{O}\left(c^{\log n}\right)$ in terms of number of Hom.Mul (tree).
- Noise budget is shrunken by many bits.

Relinearization

- Requires relinearization key (public).
- Convert from (c_0, c_1, c_2) to a new (c_0, c_1) with added noise.
- $\mathcal{O}(n \log n \log Q)$ in terms of integer add / mul mod Q.
- Introduces additive noise (negligible in Microsoft SEAL).
- Noise budget is barely shrunken.

Rotation

- Requires Galois keys (public).
- Rotates hidden message slots (1,2,3,4, …)→(3,4, …, 1,2).
- $\mathcal{O}(n \log n \log Q)$ in terms of integer add / mul mod Q.
- Introduces additive noise (negligible in Microsoft SEAL).
- Noise budget is barely shrunken.

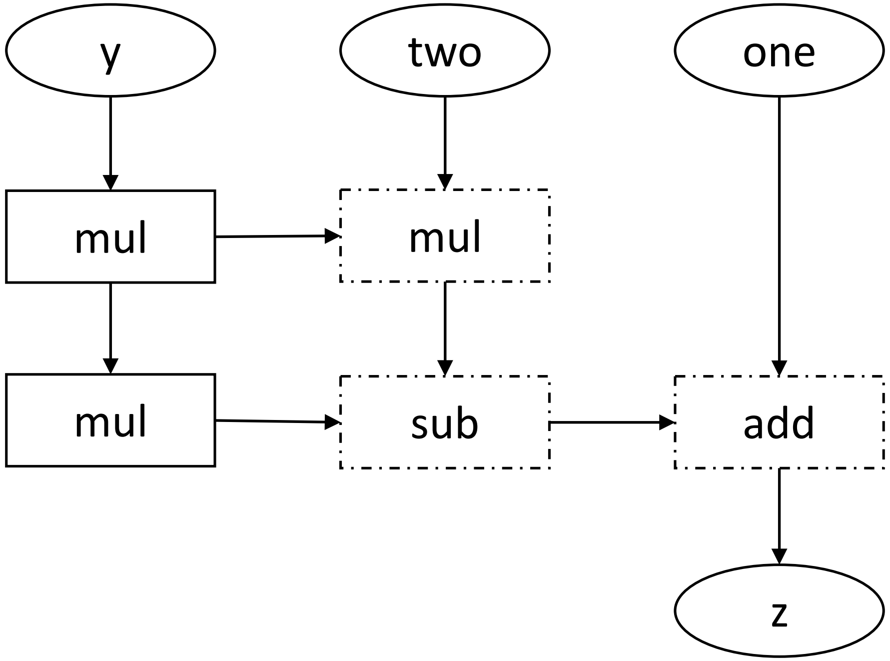
Combining Operations

- Polynomial evaluation: $z = y^4 - 2y^2 + 1$ for $y = (1,2,3,4,0,0,\cdots)$

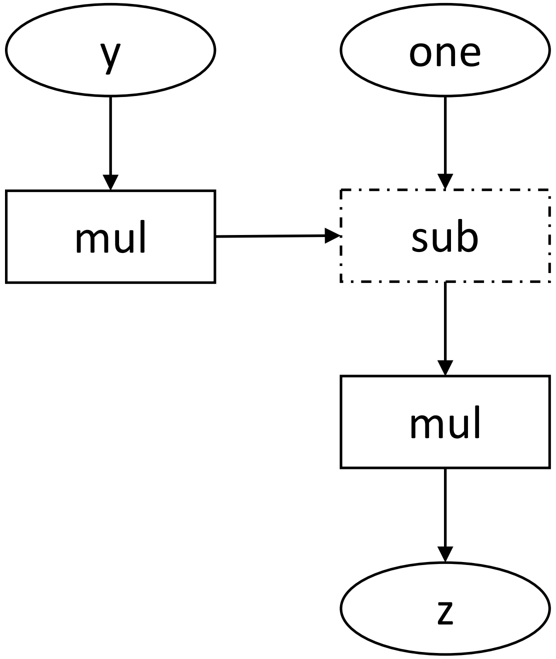
```
vector<uint64_t> y(nslots, 0ULL);
y[0] = 1ULL; y[1] = 2ULL; y[2] = 3ULL; y[3] = 4ULL;
Plaintext plain_y; batch_encoder.encode(y, plain_y);
Ciphertext encrypted_y; encryptor.encrypt(plain_y, encrypted_y);

vector<uint64_t> two(nslots, 2ULL), one(nslots, 1ULL);
Plaintext plain_two; batch_encoder.encode(two, plain_two);
Plaintext plain_one; batch_encoder.encode(one, plain_one);
Ciphertext squared_y;
evaluator.multiply(encrypted_y, encrypted_y, squared_y); // y^2
evaluator.relinearize_inplace(squared_y, relin_keys);
Ciphertext encrypted_z;
evaluator.multiply(squared_y, squared_y, encrypted_z); // y^4
evaluator.relinearize_inplace(encrypted_z, relin_keys);
evaluator.multiply_plain_inplace(encrypted_y, plain_two); // 2y^2
evaluator.sub_inplace(encrypted_z, encrypted_y); // y^4-2y^2
evaluator.add_plain_inplace(encrypted_z, plain_one); // y^4-2y^2+1
```

Circuit Optimization

$$y^4 - 2y^2 + 1$$



$$(y^2 - 1)^2$$



Defining Computation

- Linear functions: scalar, vector, and matrix addition/multiplication
- Polynomials: a k degree polynomial requires $\lceil \log k \rceil$ depth
- Equality → various algorithms in BFV, not in CKKS
- Comparison (larger or smaller than) → non-trivial
- Non-linear functions: Sigmoid, inverse, tan, etc. → approximation

Decrypt and Decode

- $R_Q^2 \rightarrow R_t$ *or* $R \rightarrow \mathbb{Z}_t^n$ *or* $\mathbb{C}^{n/2}$

```
Plaintext plain_z; decryptor.decrypt(encrypted_z, plain_z);
vector<uint64_t> z; batch_encoder.decode(plain_z, z);
```

Encryption / Decryption

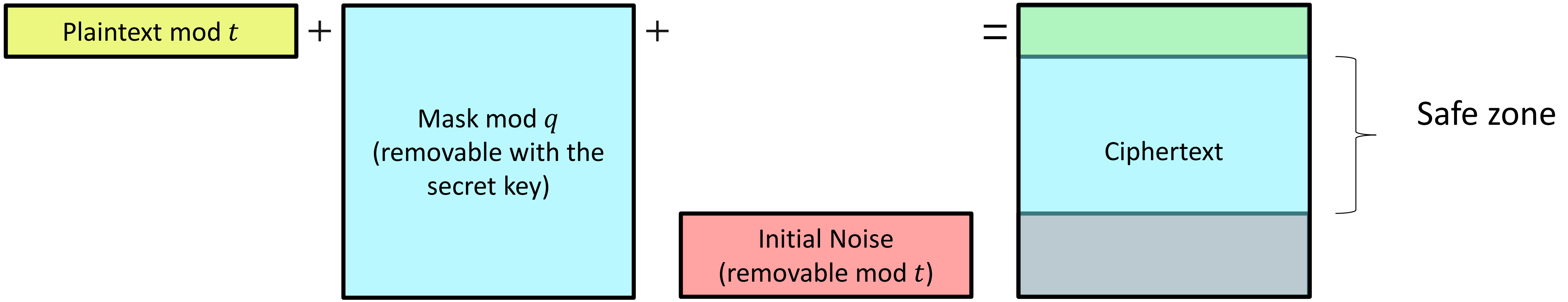
- Secret key: sample $s \leftarrow R$.
- Symmetric encryption of zero / public key:

  Sample $e \leftarrow \chi$ and $a \leftarrow R_Q$, get $(-as + e, a) \in R_Q^2$.

  This is ct or pk.
- Asymmetric encryption of zero:

  sample $u \leftarrow R$ and $e_{0,}e_1 \leftarrow \chi$, get $(pk_0 u + e_0, pk_1 u + e_1)$.
- Add plaintext to ct_0 with some modification.

Another View – Noise

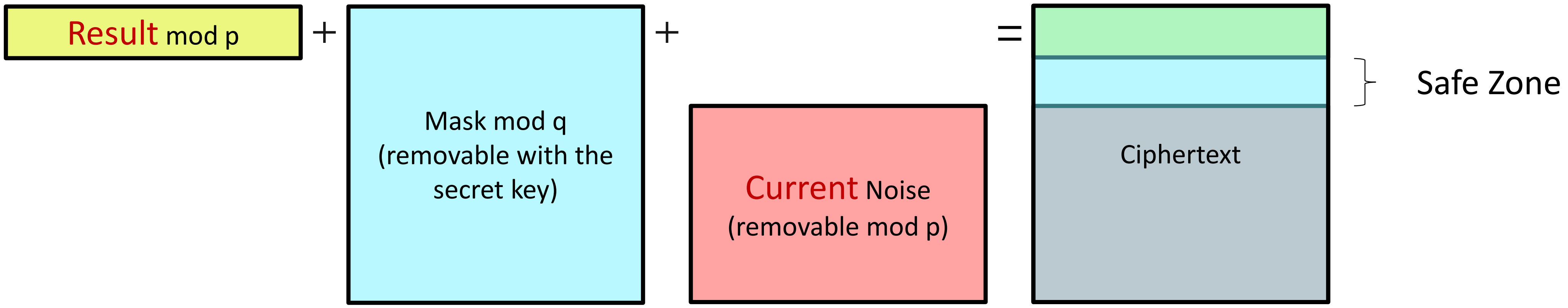


- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: bits of coefficients.

Noise in $[ct_0 + ct_1 s]_Q = \left[\frac{Qm}{t} + e_1 + eu + e_2 s\right]_Q$ can be considered Gaussian.

Initial noise (of a fresh encryption) is small in terms of coefficients' size.

Noise Growth in Computation



- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

After each level, noise increases. For more depth, increase N and q.

Terminology

- HE: general
- PHE (partially): unlimited add or mul
- FHE (fully): unlimited addition and multiplication
- SHE (somewhat): limited add and mul
- LHE (leveled): a subset of SHE, limited by levels specifically
- Bootstrapping: homomorphically refreshes a ciphertext
- FHE $\leftarrow$ SHE + Bootstrapping

Setup Security Parameters

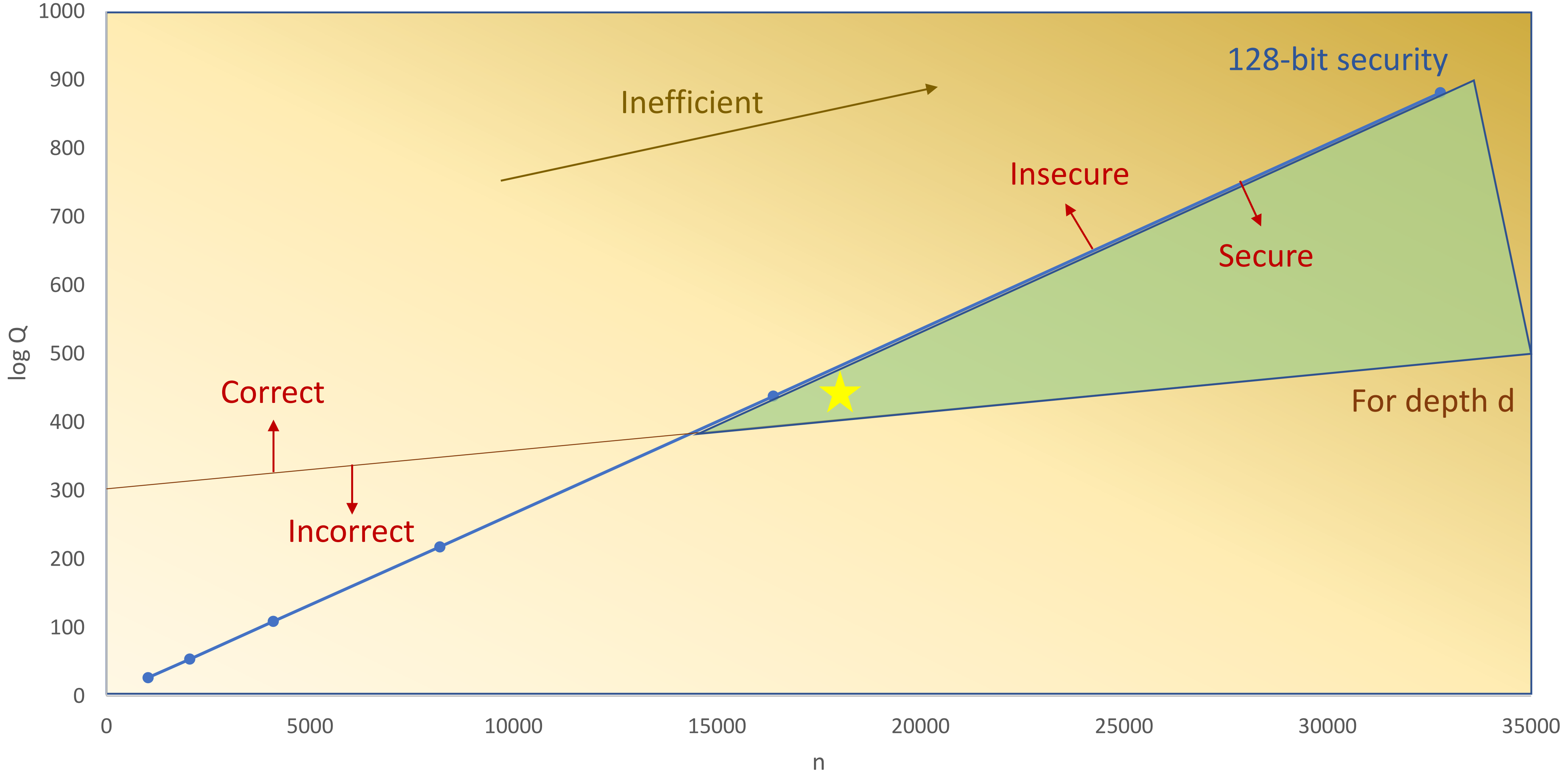
- For more depth, increase Q.
- For security, increase n.
- Homomorphic Encryption Security Standard (November 21, 2018) – homomorphicencryption.org
- Classic security (Table 1), ternary secret key, 128-bit security, suggested $(N, \log q)$ pairs:

(1024, 29) (2048, 56) (4096, 111) (8192, 220) (16384, 440) (32768, 880)

Setup Security Parameters

```
EncryptionParameters parms(scheme_type::BFV);
size_t poly_modulus_degree = 8192;
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
```

Choosing Parameters



Full-RNS Variants

- Residue Number System (RNS) – Chinese Remainder Theory (CRT)
- Given coprime moduli 3, 5, and 7, numbers from 0 to 104 can be represented as 3-tuples of their remainder divided by 3, 5, and 7.
  - 8 → (2, 3, 1), 20 → (2, 0, 6)
  - (1, 3, 0) ← 28 = 8+20 → (2+2, 3+0, 1+6) → (4, 3, 7) → (1, 3, 0)
  - (1, 0, 6) ← 55=160 = 8x20 → (2x2, 3x0, 1x6) → (4, 0, 6) → (1, 0, 6)

- Speedup integer arithmetic: big integer → multiple smaller integers.
- Choose $\mathrm{Q} = q_0 q_1 \cdots q_l p$, a product of word-sized prime numbers.

Modulus Switching

- Safely scales ciphertext mod $q_0 q_1 \cdots q_l$ to a new ciphertext mod $q_0 q_1 \cdots q_{l-1}$, reducing the hidden noise to roughly $1/q_l$.
- Noise budget is not shrunken.
- Less primes means faster computation.

- Noise-free relinearization and rotation requires p larger than q_i.

Setup Performance Parameters

- Batching requires $2n|(t-1)$.
- Fast polynomial multiplication requires $2n|(q_i-1)$ and $2n|(p-1)$.

```
parms.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree, 20));
parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, {40,40,40,50}));
```

Design A Circuit

- HE (SEAL) is a new hardware:
  - CPU excels in branching and integers modulo 2^{32} or 2^{64}
  - GPU excels in 16-, 32-, and 64-bit floating point
  - BFV excels in customized integers modulo t
  - CKKS excels in fixed precision (approximated or truncated)
- Express algorithm with modulo t or fixed precision arithmetic
- Minimize the depth of multiplications
- Limitation:
  - No conditional branching based on a ciphertext.
  - Only linear function or polynomial are native on ciphertexts.
  - No equality check in CKKS, no comparison in neither CKKS or BFV.

Encryption Parameters

1. Knowing the depth
2. Set n and $\log Q$ accordingly
3. Set $q_0, q_1, \cdots, q_l, p$, coefficient modulus
   - $\log q_0 q_1 \cdots q_l p \leq \log q$
   - p is the largest
   - BFV: as few as possible
   - CKKS: will be explained in later sessions
4. BFV only: set t, the plaintext/message modulus
5. CKKS only: set scale, explained in later sessions

Cost (Computation, Bandwidth)

- Ciphertext / messages size expansion rate is $\leq \frac{2 \log q_0 q_1 \cdots q_l}{\log t}$ in BFV.
- Computation cost in terms of word-sized integer arithmetic:
  - Addition ct-ct or ct-pt: $\mathcal{O}(ln)$
  - Multiplication ct-ct or ct-pt: $\mathcal{O}(ln \log n)$ or $\mathcal{O}(ln)$ in SEAL CKKS
  - Relinearization and rotation: $\mathcal{O}(l^2 n \log n)$
  - Modulus switching or rescaling in CKKS: $\mathcal{O}(ln)$ or $\mathcal{O}(ln \log n)$ in SEAL CKKS
- Print benchmarks from Microsoft SEAL Examples!

Practice Problems

1. Rewrite "`ct_a == ct_b ? x : y`" (select x if ct_a and ct_b decrypt to the same plaintext, select y otherwise), you can use "`Hom.Equals(ct_a, ct_b)`" that outputs an encryption of 1 if equals and an encryption of 0 otherwise.

2. Rewrite "`ct_a == ct_b`" with only addition and multiplication using BFV, suppose that the plaintext modulus t is a prime, you may ignore relinearization. (Hard question)

3. To encrypt and compute the dot product of (1, 2, 3, 4) and (2, 3, 4, 5) using BFV, choose the minimum plaintext modulus t.

4. To encrypt and compute the product of 1.1 and 1.2 using BFV, choose the minimum plaintext modulus t.