CSE208: Advanced Cryptography (FHE)

Daniele Micciancio

UCSD

Winter 2023

Section 1

Multiplication

What we have done so far

Simple LWE Encryption: private key encryption supporting

- small message modulus ($p \ll q$)
- homomorphic addition
- homomorphic multiplication by small constants
- enough to obtain public key encryption
- circular security (for small keys)

Extended LWE Encryption to support

- large message modulus ($p = q$)
- homomorphic multiplication by arbitray constants
- circular security (for arbitary keys)
- key switching

Next: Homomorphic Multiplication

Problem

Given `Enc(sk,msg[0])` *and* `Enc(sk,msg[1])`*, compute a ciphertext* `ct` *such that* `Dec(sk,ct)= msg[0]*msg[1]`

- Can this be done for our LWE encryption scheme?
- Can it be done with the help of some additional key material?
- Yes, in fact, there are multiple ways to do it
  - Nested encryption
  - Homomorphic decryption
  - Tensor product

Method 1: Nested Encryption

- msg[0],msg[1] $\in \mathbb{Z}_q$
- ct[0] = Enc(sk[0],msg[0])
- ct[1] = Enc(sk[1],msg[1])
- Multiply encryption of msg[0] by ct[1]

```
ct[0] * ct[1]
= Enc(sk[0],msg[0]) * ct[1]
= Enc(sk[0],msg[0]*ct[1])
```

- Inner multiplication:

```
msg[0]*ct[1]
= msg[0]*Enc(sk[1],msg[1])
= Enc(sk[1],msg[0]*msg[1])
```

- Final result: Enc(sk[0],Enc(sk[1],msg[0]*msg[1]))

Details

- ct[1] = Enc(sk[1],msg[1]) is a vector!
  - ct[0] = Enc(sk[0],msg[0]*I)
  - (msg[0]*I)*ct[1] = msg[0]*ct[1]
- msg[0]*Enc(sk[1],msg[1];e[1])= Enc(sk[1],msg[0]*msg[1]; msg[0]*e[1])
  - Assume msg[0] is small (e.g., ,10,1)
  - May set Enc(sk[1],msg[1])= LWE(sk[1],(q/2)*msg[1])
- Using Enc(sk[0],Enc(sk[1],msg))
  - Keep nesting?
  - Ciphertexts get larger and larger!

Key Nesting

- Recall: `Enc(S,M)= LWE(S,M)= (A,S*A + E + M)`
- Claim: Nested encryption `Enc(Z,Enc(S,M))= Enc(Z◇S,M)`

Question

For what key `Z◇S`*?*

Key Nesting

- Recall: Enc(S,M)= LWE(S,M)= (A,S*A + E + M)
- Claim: Nested encryption Enc(Z,Enc(S,M))= Enc(Z⋄S,M)

Question

For what key Z⋄S?

- S:$\mathbb{Z}$[k,n], Z:$\mathbb{Z}$[n+k,n]
- Z = (Zn,Zk) where Zn[n,n] and Zk[k,n]
- Z⋄S = [S*Zn + Zk, S] = [S,I]Z
  - Enc(Z,Enc(S,m;e);e')= Enc(Z⋄S,m; e")
  - e" = e + [S,I]e'
  - Key S needs to be small!

Nested Encryption + (Sub)Key Switching

Combine nested multiplication with key switching:

- Input keys: `Z,S`
- Evaluation key: `W = Enc(S,[S,I]Z;F)`
- Input ciphertexts:
  - `CT[0] = Enc(Z,msg[0]*I;E[0])`
  - `CT[1] = Enc(S,msg[1]*I;E[1])`
- Output: `SubkeySwitch(W,CT[0]*CT[1])= Enc(S,msg*I;E)`
  - `msg = msg[0]*msg[1]`
  - `E = msg[0]*E[1] + [S,I]*E[0]*X + F*Y` for binary matrices `X,Y`
- Key `S` needs to be small!
- Security Assumption: Standard LWE

Method 1.5: Homomorphic Decryption

- Assume both ciphertexts use the same key `S`
- Nested Encryption:
  1. Homomorphic multiplication: `Enc(S,msg[0])*CT[1]`
  2. Key Switching: Homomorphic multiplication by `[S,I]`
- Method 1: `Eval([S,I],Enc(S,msg[0])*CT[1])`
- Combine the two homomorphic multiplications:
  - Bring `[S,I]` inside the first ciphertext
  - `Enc(S,msg[0]*[S,I])* CT[1]`
- Define a new LWE encryption variant:

$$\mathtt{Enc}^{\#}(\mathtt{S},\mathtt{msg}) = \mathtt{Enc}(\mathtt{S},\mathtt{msg}*[\mathtt{S},\mathtt{I}])$$

$$\begin{aligned}&\mathtt{Enc}^{\#}(\mathtt{S},\mathtt{msg}[0]) * \mathtt{Enc}(\mathtt{S},\mathtt{msg}[1])\\ &= \mathtt{Enc}(\mathtt{S},\mathtt{msg}[0]*[\mathtt{S},\mathtt{I}]*\mathtt{Enc}(\mathtt{S},\mathtt{msg}[1]))\\ &= \mathtt{Enc}(\mathtt{S},\mathtt{msg}[0]*\mathtt{msg}[1])\end{aligned}$$

Security

$$\begin{aligned} \mathtt{Enc}^{\#}(\mathtt{S},\mathtt{msg}) &= \mathtt{Enc}(\mathtt{S},\mathtt{msg}*[\mathtt{S},\mathtt{I}]) \\ &= [\mathtt{Enc}(\mathtt{S},\mathtt{msg}*\mathtt{S}),\mathtt{Enc}(\mathtt{S},\mathtt{msg}*\mathtt{I})] \end{aligned}$$

- Circular security:
  - Can compute $\mathtt{Enc}(\mathtt{S},\mathtt{msg}*\mathtt{S}) = \mathtt{msg}*\mathtt{Enc}(\mathtt{S},\mathtt{S})$ without knowing $\mathtt{S}$
  - Problem: $\mathtt{msg}*(-\mathtt{I},0)$ reveals $\mathtt{msg}$!
  - Solution: $\mathtt{Enc}(\mathtt{S},0) + \mathtt{msg}*\mathtt{Enc}(\mathtt{S},\mathtt{S})$

Theorem

$\mathtt{Enc}$ *is secure under the LWE assumption*

Remarks

- Second encryption scheme can be chosen arbitrarily

$$\mathtt{Enc}^{\#}(S, m_0) * \mathtt{Enc}(S, m_1) = \mathtt{Enc}(S, m_0 m_1)$$
$$\mathtt{Enc}^{\#}(S, m_0) * \mathtt{Enc}^{\#}(S, m_1) = \mathtt{Enc}^{\#}(S, m_0 m_1)$$

- No need for key switching
  - Product $\mathtt{Enc}(S, m_0 m_1)$ uses the same key as the input
  - Key S does not have to be small
  - No evaluation key!
- Enc is a homomorphic encryption scheme supporting
  - Ciphertext addition
  - Ciphertext multiplication
  - without any evaluation key!
- Too good to be true?

Error growth

- $\mathsf{Enc}^{\#}(m_0; E_0) * \mathsf{Enc}^{\#}(m_1; E_1) = \mathsf{Enc}^{\#}(m_0 m_1; E)$
  - Error: $E \approx m_0 * E_1 + E_0 * X$
- Multiplying many ciphertexts
  - `CT[i]` = $\mathsf{Enc}^{\#}(m_i; E_i)$
  - Assume $m_i \in 0, 1$
  - Given `CT[1],...,CT[k]`
  - Goal: compute `CT[1]*...*CT[k]` = $\mathsf{Enc}^{\#}(\prod_i m_i)$
- How? Several options (multiplication is associative):
  - Left to right multiplication chain
  - Right to left multiplication chain
  - Binary tree (minize circuit depth)

Question

What order is best?

Arithmetic and Boolean operations

- Addition
  - Can add polynomially many ciphertexts
  - Error grows by polymomial factor (e.g., $O(log(n))$ bits)
- Multiplication
  - Assume **binary** message space
  - Can multiply polynomially many ciphertexts in a **chain**
  - Error grows by polymomial factor (e.g., $O(log(n))$ bits)

Arithmetic and Boolean operations

- Addition
  - Can add polynomially many ciphertexts
  - Error grows by polymomial factor (e.g., $O(log(n))$ bits)
- Multiplication
  - Assume **binary** message space
  - Can multiply polynomially many ciphertexts in a **chain**
  - Error grows by polymomial factor (e.g., $O(log(n))$ bits)
- Bit operations:
  - $m_0, m_1 \in \{0, 1\}$
  - $m_0 \wedge m_1 = m_0 \cdot m_1$
  - $\neg m_0 = 1 - m_0$
  - $m_0 \vee m_1 = \neg(\neg m_0 \wedge \neg m_1)$
- Conditional: $(b, m_0, m_1) \mapsto m_b$
  - $m_b = (1 - b) \cdot m_0 + b \cdot m_1$
- Arbitrary log-depth boolean circuits

Method 2: Tensor and Key Switch

- Why? Efficiency! Allows SIMD operations using polynomial rings
- Ciphertext as a function
  - f_C(S)= Dec'(S,C)= [S,I]C
  - f_C is linear in [S,I]
- Product ciphertext C = C0*C1
  - Goal: Dec'(S,C)= Dec'(S,C0)*Dec'(S,C1)
  - $f_{C0,C1}$(S)= Dec'(S,C0)*Dec'(S,C1) is bilinear in [S,I]
- Tensor product: Z = [S,I]⊗[S,I] = [S⊗S,S,S,I]
  - Any bilinear function of [S,I] is linear in Z
  - C = C0⊗C1 decrypts to $m_0 \cdot m_1$ under Z

Mixed product property

Theorem

For any A, B, X, Y,

$$(A \otimes B) \cdot (X \otimes Y) = (A \cdot X) \otimes (B \cdot Y)$$

Mixed product property

Theorem

For any A, B, X, Y,

$$(A \otimes B) \cdot (X \otimes Y) = (A \cdot X) \otimes (B \cdot Y)$$

$$\begin{aligned}([S, I] \otimes [S, I]) \cdot (C_0 \otimes C_1) &= ([S, I]C_0) \otimes ([S, I]C_1) \\ &= (X_0 + E_0) \otimes (X_1 + E_1)\end{aligned}$$

Result: $X_0 \otimes X_1 + X_0 \otimes E_1 + E_0 \otimes X_1 + E_0 \otimes E_1$

- Assume scalar messages: $x_0 \otimes x_1 = x_0 \cdot x_1$
- Messages must be encoded: $x_i = \frac{q}{p} m_i$

Encoding issues

- Encode scalar messages: $x_i = \frac{q}{p} m_i$
- Product: $(x_0 + e_0)(x_1 + e_1) = x_0 x_1 + x_0 e_1 + e_0 x_1 + e_0 e_1$
- Issues:
  - Error terms $x_0 e_1 + e_0 x_1 = \frac{q}{p}(m_0 e_1 + e_0 m_1)$ are too large
  - Main term $x_0 x_1 = (q/p)^2 m_0 m_1$ is not properly encoded
- Solutions:
  - **Modular arithmetics**: assume $\gcd(q, p) = 1$, and multiply result by $p \pmod q$
  - **Modulus lifting**: Compute the product modulo q^2, and then switch to smaller modulus q

Modular arithmetics

- Compute $c = p \cdot c_0 \otimes c_1 \pmod{q}$
- Output c decrypts (under $\mathsf{sk} \otimes \mathsf{sk}$) to

$$p(x_0 + e_0)(x_1 + e_1) = \frac{q}{p}(-q m_0 m_1) + p e_0 e_1$$

Modular arithmetics

- Compute $c = p \cdot c_0 \otimes c_1 \pmod{q}$
- Output c decrypts (under sk$\otimes$sk) to

$$p(x_0 + e_0)(x_1 + e_1) = \frac{q}{p}(-qm_0m_1) + pe_0e_1$$

- Assume $q = -1 \pmod{p}$
  - Error growth: $\beta \mapsto p\beta^2$
- Arbitrary q, p
  - Multiply result by $(-q)^{-1} \pmod{p}$
  - Error growth: $\beta \mapsto p^2\beta^2$
- Modulus switching can be used to reduce β to a fixed polynomial $\sigma = \|s\|_1 = O(n)$, and substantially slow down the error growth

Modulus Lifting

- Compute $c = p \cdot c_0 \otimes c_1 \pmod{q^2}$
- Assume key $\|s\|_1 < \sigma$ has small entries
- Analyze the relative error: $c_i = \mathsf{Enc}(m_i; (q/p)e_i)$

Theorem

The product $p(c_0 \bmod q) \otimes (c_1 \bmod q)$ *is an encryption of* $m_0 m_1 \pmod{p}$ *under key* $s \otimes s \pmod{q^2}$ *with error* $(q^2/p)e$

$$e \leq 3e_0e_1 + \frac{p}{2}(\sigma + 1)(e_0 + e_1)$$

Relative error growth

- Fixed polynonials $\beta \approx \sqrt{n}$, $\sigma = \|s\|_1 \approx O(n^{1.5})$
- Modulus lifting error growth
  - relative error: input $(q/p)e_i$, output $(q^2/p)e$
  - assume $|e_i| < \epsilon$
  - output (multiplication) error $\approx p\sigma\epsilon$
- After L levels of multiplications, error $\approx (p\sigma)^L\epsilon < 1$
- Input ciphertext modulus must be $q \approx (p\sigma)^L$
  - Better than modular arithmetics approach $q > (p\beta)^{2^L}$
  - Similar growth to modular arithmetics + modulus switching

Tensoring + Key Switching

- Both methods produce a ciphertext under key $[S \otimes S, I \otimes S, S \otimes I]$
- For scalar messages $I = [1]$ and $I \otimes S = S \otimes I = S$
- Can use subkey switching from $[S \otimes S, S]$ to S
- Evaluation key: $\mathsf{Enc}(S, S \otimes S)$
- Security:
  - Does not follow from circular security of LWE
- Using standard LWE:
  - Evaluation key $\mathsf{Enc}(Z, S \otimes S)$
  - Use a sequence of keys $S_0, ..., S_L$, one for each multiplicative level of circuit/computation
  - Can you still use subkey switching?

Arithmetic computations using Tensor products

- Message encoding: $(q/p)m$
- Plaintext arithmetic modulo p (both addition and multiplication)
- Error grows with multiplicative depth of the circuit
- Use small key $\|s\|_1 < \sigma$ to use modulus switching and slow down error growth
- Error at depth L: $\approx (p\sigma)^L < q$
  - $L = O(\log n)$: $q = n^{O(\log n)}$
  - $L = poly(n)$: $q = 2^{poly(n)}$
- Impact of modulus:
  - Efficiency: running time $poly(\log q)$
  - Security: requires hardness of of approximating lattice problems within $\gamma \approx q/\beta$