Section 8

FHE!!
Bootstrapping

- Given (1-hop) \((\text{Gen}, \text{Enc}, \text{Dec}, \text{Eval})\) supporting functions
  \[ f_{c,c'}(sk) = \text{Dec}(sk,c) \text{ nand } \text{Dec}(sk,c') \]

- Define (multi-hop) FHE scheme with \(\text{Func} = \{ \text{nand} \}\)

\[
\text{Gen}'() = (sk, pk) \leftarrow \text{Gen}()
\]
\[
ek \leftarrow \text{Enc}(pk, sk)
\]
\[
\text{return } (sk, (pk, ek))
\]

\[
\text{Enc}'((pk, ek), m) = \text{Enc}(pk, m)
\]

\[
\text{Eval}'((pk, ek), \text{nand}, c, c')
\]
\[
= \text{EvalC}(pk, f_{c,c'}, ek)
\]
LWE Homomorphic Encryption

- Goal: homomorphic evaluation of
  \[ f_{c,c'}(sk) = Dec(sk, c) \text{ nand } Dec(sk, c') \]

- LWE-based cryptosystem
  - Supports bounded depth addition and multiplication
  - Bit operations: \( x \text{ nand } y = 1 - (1-x) \cdot (1-y) \)

- Key Switching
  \[ ek[i] = Enc'(sk', sk[i]) \]
  \[ KeySwitch(ek, (a[], b)) = Const(b) - \sum_i CMul(a[i], ek[i]) \]

- Homomorphically evaluate of \( Dec'(a,b) = b - Sa \)
Key switching only computes the linear part of Dec

We also need to round the result to decode(b - Sa)

Is this really needed?

Yes, b - Sa = (q/p)m + e

Key switching gives a noisy encryption of (q/p)+e

Without rounding, noise keeps getting bigger

Questions

Can we express rounding as a polynomial function (mod q)?

What is the degree of the polynomial?
Error growth and bounded computation

We have seen two methods to multiply ciphertexts:

- **Tensor products**
  - error growth $\sim \beta \rightarrow \beta \sigma$
  - can evaluate arbitrary circuits with **multiplicative depth** $L$
  - even for $L = \log n$, requires superpolynomial modulus $q > \sigma^L \approx n^{O(\log n)}$

- **Nested Encryption / Homomorphic Decryption**
  - asymmetric error growth: $(m_0, e_0) \times (m_1, e_1) \rightarrow m_0 e_1 + e_0 \beta$
  - can evaluate arbitrary multiplication **chains** of $L$ fresh encryptions of **binary** messages
  - even for large $L$, polynomial modulus $q \approx L \beta^2$ is enough
Roadmap

For each multiplication method

1. Describe/analyze a bootstrapping algorithm
2. Homomorphically evaluate the algorithm using an appropriate cryptographic data structure (encrypted accumulator)
3. Implement the cryptographic data structure using LWE
Cryptographic accumulators

- **Cryptographic Data Structure** $\text{ACC}[v]$
  - Holds a value $v \in V$ in encrypted form
  - Input Encryption scheme: $\text{Enc}'$
  - Output Encryption scheme: $\text{Enc}''$

- **Operations on** $\text{ACC}[v]$
  - Given $\text{Enc}'(x)$, update $\text{ACC}[v] \rightarrow \text{ACC}[f(v,x)]$
  - Given $\text{ACC}[v]$, output $\text{Enc}''(f(v))$

- **Bootstrapping:**
  - Bootstrapping key: $\text{Enc}'(s)$
  - Final output: $\text{Enc}''(m)$
Assume $p = 2, m \in \{0, 1\}$

Decryption Algorithm:

- Input: $a[1..n] \in \mathbb{Z}_q^n, b \in \mathbb{Z}_q$
- Secret key: $s[1..n] \in \mathbb{Z}^n$
- Compute $d = b - \sum_i a[i]s[i] + (q/4) \pmod{q}$
- Round $d$ to $\text{MSB}(d) = \lfloor 2d/q \rfloor$

Homomorphic Computation:

- Given $\text{Enc}(s[i])$
- Compute $\text{Enc}(\text{MSB}(d))$

Simplifying assumption:

- $s[i] \in \{0, 1\}$
- without loss of generality using $(a, 2a, 4a, ...)$
Ripple-carry addition

- Standard schoolbook method
  - using binary digits
  - add $n$ numbers at a time
  - carry in $\{0, \ldots, n\}$

- Input digits are encrypted
Ripple-carry accumulator

- **Parameters:**

- Message space $V = \{v', \ldots, v''\}$

- Input: $\text{Enc'}(x) = \text{Enc#}(x)$

- Output: $\text{Enc''}(x) = \text{LWE}(x)$

- $\text{ACC}[x] = (\text{Enc''}("x=v") : v \in V)$

  - $\text{Init}(v) = \text{ACC}[v]$
  - Function application: $f(\text{ACC}[v]) = \text{ACC}[f(v)]$
  - Selection:
    $\text{Enc'}(b) ? \text{ACC}[v0] : \text{ACC}[v1] = \text{ACC}[b?v0:v1]$
  - Output: $p(\text{ACC}[v]) = \text{Enc''}(p(b))$
Bootstrapping algorithm

\[ \begin{align*}
  b + q/4 &= \sum_j 2^j b[j] \\
  a[i] &= \sum_j 2^j a[i,j] \\
  \text{ACC} &\leftarrow \text{ACC}[0] \\
  \text{for } h = 0..k-1 \\
  \quad \text{ACC}[x] &\leftarrow f(\text{ACC}[x]) \text{ where } f(x) = (x/2) + b[h] \\
  \quad \text{forall } i,j \\
  \quad \quad \text{if } (a[i,j] = 1) \\
  \quad \quad \quad \text{ACC}[x + s[i]] &\leftarrow \text{Enc}'(s[i]) \ ? \ \text{ACC}[x] : \ \text{ACC}[x+1] \\
  \text{return } (\text{even(ACC}[x]]) = \text{Enc}''(\text{even(x)})
\end{align*} \]
Carry-save accumulator

- Parameters: bit length $k$

- $\text{ACC}[x] = (x_0, x_1)$
  - $x = x_0 + x_1 \pmod{2^k}$
  - $x_0[0, \ldots, k-1]$ and $x_1[0, \ldots, k-1]$
  - redundant representation

- Operations:
  - add $y$ to ACC
  - compute $\text{MSB}(\text{ACC})$
### Carry-save addition

Add($\text{ACC}(x_0, x_1), y$):

\[
x_0'[i] = (x_0[i] + x_1[i] + y[i]) \mod 2 \\
x_1'[i+1] = (x_0[i] + x_1[i] + y[i] > 1)\]

\[\text{return } \text{ACC}(x_0', x_1')\]
MSB computation

- Standard MSB computation
  - addition $x_0 + x_1$ with carry propagation
  - $O(\log(k))$ depth circuit where $k = \log(q)$

- Can also add in $\log(k)$ depth
  - Compute both $\text{MSB}(\text{ACC})$ and $\text{MSB}(\text{ACC}+1)$
  - $\text{ACC}[k]$: $k$-bit accumulator
  - Recursive algorithm: split
    $\text{ACC}[k] = (\text{HiACC}[k/2], \text{LoACC}[k/2])$

$\text{MSBs}(\text{ACC}=(\text{HiACC}, \text{LoACC}))$:
  parallel:
    $hi[0,1] = \text{MSBs}(\text{HiACC})$
    $lo[0,1] = \text{MSBs}(\text{LoACC})$
    $out[0] = hi[lo[0]]$
    $out[1] = hi[lo[1]]$

$\text{return } out$
Bootstrapping algorithm

\[
\begin{align*}
\text{ACC}[0] &= b + q/4 \\
\text{for } i &= 1 \ldots n \\
\quad \text{ACC}[i] &= s[i] \times a[i] \\
\text{ACC} &= \text{Sum}(\text{ACC}[0], \ldots, \text{ACC}[n]) \\
\text{return } \text{MSB}(\text{ACC}) \\
\end{align*}
\]

\[
\begin{align*}
\text{Sum}(\text{ACC}[0..n]) \\
\text{if } n &= 0 \\
\quad \text{then return } \text{ACC}[0] \\
\text{else } h &= n/2 \\
\quad \text{ACC0} &= \text{Sum}(\text{ACC}[0..h-1]) \\
\quad \text{ACC1} &= \text{Sum}(\text{ACC}[h..n]) \\
\quad (x0, x1) &= \text{ACC1} \\
\quad \text{return } (\text{ACC0} + x0) + x1
\end{align*}
\]
Summary

Bootstrapping functions can be computed by

1. $O(n \log q)$-long sequence of multiplications, or
2. $\log(n) + \log \log(q)$-depth arithmetic circuits

Error growth:

1. Using $\text{LWE} \circledast$: final error $\approx O(n) \cdot \beta$
2. Using $\text{LWE} \otimes$: final error $\approx \sigma^{\log n + \log \log q} = \sigma^{O(\log n)}$

Parameters $\beta(n), \sigma(n)$: fixed polynomials in $n$

Modulus:

1. polynomial modulus $q(n) \approx O(n)\beta = n^{O(1)}$
2. quasipolynomial $q(n) = n^{O(\log n)}$
Summary (security)

- Hardness of lattice problems within factor $\gamma \approx q/\beta$
  1. $\text{LWE} \odot$: polynomial $\gamma = n^{O(1)}$
  2. $\text{LWE} \otimes$: quasipolynomial $\gamma = n^{O(\log n)}$

- Circular security assumption
  - Needed by tensor product multiplication / keyswitching
  - Needed to apply bootstrapping
  - Not needed for leveled homomorphic encryption
Summary (security)

- Hardness of lattice problems within factor $\gamma \approx q/\beta$
  - LWE $\circ$: polynomial $\gamma = n^{O(1)}$
  - LWE $\otimes$: quasipolynomial $\gamma = n^{O(\log n)}$

- Circular security assumption
  - Needed by tensor product multiplication / keyswitching
  - Needed to apply bootstrapping
  - Not needed for leveled homomorphic encryption

Question

Remove circular security assumption:
- Can you build (unbounded) FHE from standard LWE?
- Can you build (unbounded) linearly homomorphic HE?
Efficiency

- Main security parameter $n > 100$ (typically, $n \approx 1000$)
- Modulus $q(n) < 2^n$ has bitsize $\log q < n$
- Assume 1GHz, arithmetic operations modulo $q$
- Bootstrapping: homomorphically evaluate decryption algorithm (once or twice per gate)

**Question**

*Can you estimate the cost of a single FHE operation?*