CS 6501 Week 13: Advanced Lattice-Based Primitives

So far, we have shown how to build symmetric crypto and public-key crypto from standard lattice assumptions (e.g., BIS and LWE).

But it turns out, lattices have much additional structure ⇒ enable many new advanced functionalities not known to follow from many other standard assumptions (e.g., discrete log, factoring, pairing, etc.)

We will begin by studying **fully homomorphic encryption (FHE)**.  
⇒ encryption scheme that supports arbitrary computation on encrypted data (very useful for outsourced computation)

**Abstractly:** given encryption of e under some public key, can we derive from that an encryption of f(x) for an arbitrary function f?

- So far, we have seen examples of encryption schemes that support one type of operation (e.g., addition) on ciphertexts
  - ElGamal encryption (in the exponent): homomorphic with respect to addition [1983]
  - Regev encryption: homomorphic with respect to addition
  - For FHE, need homomorphism with respect to two operations: addition and multiplication

**Major open problem in cryptography (dates back to late 1970s)** - first solved by Stanford student Craig Gentry in 2009

L⇒ revolutionary lattice-based cryptography!

L⇒ very surprising this is possible: encryption needs to "scramble" messages to be secure, but homomorphism requires preserving structure to enable arbitrary computation

**General blueprint:** 1. Build somewhat homomorphic encryption (SWHE) — encryption scheme that supports bounded number of homomorphic operations
    - Bootstrapping SWHE to FHE (essentially a way to "refresh" ciphertext)

**Focus will be on building SWHE (has all of the ingredients for realizing FHE)**

L⇒ In particular, will present Gentry-Sahai-Waters (GSW) construction (conceptually simplest scheme, though not the most concretely efficient)

"3rd generation of FHE"

**Starting point:** Regev's encryption scheme:

- **Setup (pk):** 
  \[ \hat{A} \in \mathbb{Z}_{q}^{m \times n}, \quad s \in \mathbb{Z}_{q}, \quad e \leftarrow \mathcal{R}_{\mathbb{Z}_{q}^{n}}, \quad s = \left[ \hat{s} \hat{A} + e \right] \in \mathbb{Z}_{q}^{m}, \quad \text{Output } pk = \hat{A} \text{ and sk} = s \]

- **Encrypt (pk, m):** Write \[ c = \hat{A}r + \mu \cdot \hat{s} \cdot 1_{m \times m} \]
  \[ \text{I}_{m \times m} = \left( \begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array} \right) \]

- **Decrypt (sk, c):** Write \[ sk = s \] Compute \[ s^{T}c \] and output 0 if \[ |s^{T}(c_{0})| < \frac{1}{2} \] and 1 if \[ |s^{T}(c_{0})| > \frac{1}{2} \]

**Correctness:** \[ s^{T}C = s^{T}AR + \mu \cdot \hat{s} \cdot \frac{1}{2} \cdot s^{T}I_{m \times m} \]

\[ = e^{T}R + \mu \cdot \hat{s} \cdot \frac{1}{2} \cdot s^{T} \]

\[ \approx \mu \cdot \hat{s} \cdot \frac{1}{2} \cdot s^{T} \]

**Observe:** the vector \( s \) (i.e., the secret key) is an approximate left eigen-vector of the matrix \( C \) (i.e., the ciphertext) with associated eigenvalue \( \mu \cdot \frac{1}{2} \) (i.e., the "encoded" message)

**Security:** Same as prov for Regev encryption (two hybrids: LWE, then LHL)
Observe: We can pad \( A \) with rows of all-zeros so it is a square matrix (over \( \mathbb{Z}_q^{mn} \)) and pad \( s \) accordingly as well.

For the decryption, we justprint the message in the first \( (n+1) \) components.

Thus, correctness and security follows as before (scheme has not changed), and the message is simply the “noisy” eigenvalue associated with \( s \) (the “noisy” eigenvalue)

**Why is this view useful?** Because eigenvalues add and multiply:

- Suppose \( \lambda_1 \) is a (left) eigenvalue of \( C_1 \) with associated left-eigenvector \( s \) \[ \text{Then: } s^T(C_1 + C_2) = s^T C_1 + s^T C_2 = \lambda_1 s^T + \lambda_2 s^T = (\lambda_1 + \lambda_2) s^T \]

- Suppose \( \lambda_2 \) is a (left) eigenvalue of \( C_2 \) with associated left-eigenvector \( s \) \[ s^T C_1 C_2 = \lambda_1 s^T C_2 = \lambda_1 \lambda_2 s^T \]

Unfortunately, this intuition does not directly translate to our setting:

**Correctness:** \( s^T C = x \cdot [\frac{\lambda}{2}] \cdot s^T + e^T R \)

**Addition:** \( s^T (C_1 + C_2) = (x_1 \cdot [\frac{\lambda}{2}] + s^T R_1 + x_2 \cdot [\frac{\lambda}{2}] + s^T R_2) \]

\[ = ((x_1 + x_2) \cdot [\frac{\lambda}{2}] + s^T (R_1 + R_2)) \quad \text{works as long as } R_1 + R_2 \text{ is small! (As long as } B < q, \text{ this will be OK)}

**Multiplication:** \( s^T C_1 C_2 = (x_1 \cdot [\frac{\lambda}{2}] + s^T R_1)C_2 = x_1 \cdot [\frac{\lambda}{2}] \cdot s^T C_2 + s^T C_2 R_2 \]

\[ = x_1 \cdot [\frac{\lambda}{2}] \left( x_2 \cdot [\frac{\lambda}{2}] + s^T R_2 \right) + s^T C_2 R_2 \]

not quite what we wanted due to the message encoding, but should be fixable...

**Does Correctness fail for multiplication?**

Need a new trick: the gadget matrix \( G \) (e.g., the power-of-two matrix)

One issue above is noise growth (and bit-decomposition is effective way of generating short matrices).

The GSID FHE scheme:

**Setup**: \( A \in \mathbb{Z}_q^{mn} \) \( e \in \mathbb{Z}_q \) \[ s = \frac{A}{e^T A + e} \in \mathbb{Z}_q^{mn} \]

output \( \text{pk} = A \) and \( \text{sk} = s \)

**Encrypt (pk, m):** Write \( \text{pk} = A \in \mathbb{Z}_q^{mn} \) and sample \( R \in \{0,1\}^{mn} \)

Output \( C = AR + m \cdot G \)

**Decrypt (sk, C):** Write \( sk = s \). Compute \( s^T C \) and output 0 if \( |s^T C| < \frac{q}{4} \) and 1 if \( |s^T C| > \frac{q}{4} \)

**Correctness:** \( s^T C = s^T AR + m \cdot s^T G = m \cdot s^T G + e^T R \)

By construction of \( G \), \( [s^T G]_m = \Delta \log \frac{1}{\epsilon} \), so if \( e^T R \ll \frac{q}{4} \), then correctness goes through.
**Conclusion**: If we want to support circuits of multiplicative depth \( d \), we need to choose \( g = m^{o(d)} \) to accommodate the multiplications. Observe that in this case, \( \log g = O(d \log m) \), so the number of bits in the ciphertext scales linearly with the depth of the circuit. [Note: generally, there is a bit of flexibility when choosing lattice parameters]

Semantic security follows by same argument as Regev. Homomorphic operations possible by structure of gadget matrix!

From SWHE to FHE. The above construction requires imposing an a priori bound on the multiplicative depth of the computation.

To obtain fully homomorphic encryption, we apply Gentry’s brilliant insight of bootstrapping.

**High-level idea.** Suppose we have SWHE with following properties:

1. We can evaluate functions with multiplicative depth \( d \).
2. The decryption function can be implemented by a circuit with multiplicative depth \( d' < d \).

Then, we can build an FHE scheme as follows:

- Public key of FHE scheme is public key of SWHE scheme and an encryption of the SWHE decryption key under the SWHE public key.
- We now describe an ciphertext-refreshing procedure:

  - For each SWHE ciphertext, we can associate a “noise” level that keeps track of how many more homomorphic operations can be performed on the ciphertext (while maintaining correctness).
  - For instance, we can evaluate depth-\( d \) circuits on fresh ciphertexts; after evaluating a single multiplication, we can only evaluate circuits of depth-\( (d+1) \) and so on...

  - The refresh procedure takes any valid ciphertext and produces one that supports depth-\( (d-d') \) homomorphism; since \( d > d' \), this enables unbounded (i.e., arbitrary) computations on ciphertexts.

**Idea**: Suppose \( C_{x} = \text{Encrypt}(pk,x) \).

Using the SWHE, we can compute \( C_{fx} = \text{Encrypt}(pk,f(x)) \) for any \( f \) with multiplication depth up to \( d \).

Given \( C_{x} \), we first compute:

\[
C_{x} = \text{Encrypt}(pk, C_{x}) \quad \text{[strictly speaking, encrypt bit by bit]}
\]

This is a fresh ciphertext so we can perform operations of depth up to \( d \) on \( C_{x} \). Since the public key includes a copy of the decryption key \( (C_{x}) \), we can homomorphically evaluate the decryption function:

\[
C_{x} = \text{Encrypt}(pk, C_{x}) \quad \text{Encrypt}(pk, \text{Decrypt}(sk,C_{x})) = \text{Encrypt}(pk,x)
\]

This is a new encryption of \( x \), and we can continue performing homomorphic operations on \( m \) (of depth \( d-d' \)).
Bootstrapping is a general technique that converts any \( \text{SWHE} \) that can evaluate its own decryption function (plus a little more) into an \( \text{FHE} \) scheme. Transformation requires additional circular security assumption (namely, that it is \( \text{OK} \) to publish an encryption of the scheme's own public key. \[ \text{The GSW scheme supports bootstrapping -- decryption is a threshold inner product; choose parameters carefully] \]

**Open problem**: Build \( \text{FHE} \) from \( \text{LWE} \) (or another standard assumption) without the circular security assumption.

The GSW homomorphic operations have a lot of applications. We will describe three of them in the remaining weeks of this course: homomorphic signatures, attribute-based encryption, and non-interactive zero-knowledge.

**Homomorphic signatures**: Analog of homomorphic encryption for signatures

- given signature \( \sigma \) on input \( x \), can compute signature \( \sigma_y \) on any function evaluation \( f(x) \) where \( \sigma_y \) verifies with respect to function \( f \) and value \( f(x) \)
- useful for authenticating computations (eg, cloud provider can prove that performed a particular computation correctly on signed data)

**Syntax**:

\[
\text{Setup}(1^\lambda) \rightarrow (sk, vk) : \text{Outputs a signing key } sk \text{ and a verification key } vk
\]

\[
\text{Sign}(sk, x) \rightarrow \sigma_x : \text{Output a signature on a message } x
\]

\[
\text{Eval}(vk, \sigma_x, f) \rightarrow \sigma_{f(x)} : \text{Takes a signature on } x \text{ and a function } f \text{ and outputs a signature on } f(x)
\]

\[
\text{Verify}(vk, f, y, \sigma_y) \rightarrow 0/1 \text{ : Checks whether signature } \sigma_y \text{ is a signature on value } y \text{ with respect to function } f
\]

**Correctness**:

\[
(\ sk, \ vk ) \gets \text{Setup}(1^\lambda)
\]

\[
\sigma_x \gets \text{Sign}(sk, x) \implies \text{Verify}(vk, f, y, \sigma_y) = 1
\]

\[
\sigma_{f(x)} \gets \text{Eval}(vk, \sigma_x, f)
\]

*(One-Time) Unforgeability*:

[Diagram]

adversary wins if \( y \neq f(x) \) but \( \text{Verify}(vk, f, y, \sigma_y) = 1 \).

**Intuitively**: the adversary can always produce new signatures (by using the homomorphic properties of the underlying signature scheme), but cannot produce a new signature that does correspond to a valid computation on the signatures it is given.

**Compactness**: signatures are “short” (depend essentially on the size of the output and the depth of the circuit):

\[
|\sigma_{f(x)}| = \text{poly}(\lambda, d)
\]

in particular, if we compute a signature over a large database (ie, many signatures), the resulting signature that authenticates the computation can still be short.
Starting point: Recall GPV signatures (hash and sign)
\[ \text{vk}: \text{A}, \quad \text{sk}: t \text{da} \]

Signature on message \( m \) is a short vector \( u \in \mathbb{Z}_q^m \) such that \( Au = H(m) \) (modeled as random oracle)
\( t \) can view this as "target vector"

For homomorphic signatures, we will sign bit-by-bit. Suppose we are signing \( t \)-bit strings.
\[ \text{vk}: \text{A}_1, \text{V}_1, \ldots, \text{V}_t \in \mathbb{Z}_q^n \]
\[ \text{sk}: \text{tda} \]

To sign an input \( x \in \{0,1\}^t \), we will sample short \( U_1, \ldots, U_t \in \mathbb{Z}_q^n \)
\[ \text{AU}_i = V_i + x_i G \]

target matrix like in GPV
signatures (notice that this is uniform since \( V_i \in \mathbb{Z}_q^n \) and we all only ever sign a single message)

Signing directly corresponds to preimage sampling (using trapdoor for \( A \)).

Homomorphic operations are defined exactly as in GSW FHE:

- **GSW Ciphertext:** \( C = AR + x_i G \)
- **Encryption randomness** (publicly known)
- **Signature:** \( \text{AU}_i = V_i + x_i G \)

Suppose we have
\[ V_i = \text{AU}_i + x_i G \]
\[ V_z = \text{AU}_z + x_z G \]
\[ \Rightarrow V_i + V_z = A (U_i + U_z) + (x_i + x_z) G \]
\[ = \text{sum of } x_i, x_z \]
\[ \Rightarrow (U_i, U_z) \text{ can be viewed as a signature on the sum } x_i + x_z \text{ with respect to the public component } V_i + V_z \]

For multiplication, we use the analog of GSW multiplication:
\[ V_i = \text{AU}_i + x_i G \]
\[ V_z = \text{AU}_z + x_z G \]
\[ \Rightarrow V_i G^{\perp} (V_z) = A U_i G^{\perp} (V_z) + x_i A U_z + x_z G \]
\[ = \text{signature on } x_i, x_z \text{ depends only on public parameters} \]
\[ \Rightarrow \text{product of signatures} \]
\[ \Rightarrow \text{norm of signature grows by multiplicative factor } O(n) \]

\[ U_i, x_i, V_z, \text{ which are known to the evaluator} \]

**Summary:**
Suppose \( V_i = \text{AU}_i + x_i G \)
\[ \vdots \]
\[ V_z = \text{AU}_z + x_z G \]
\[ \Rightarrow \text{given } V_i, \ldots, V_z, \text{ and circuit } C, \text{ can compute } V_c \]
\[ \Rightarrow \text{using homomorphic evaluation procedure described above} \]
\[ \text{given } V_i, \ldots, V_z, \text{ signatures } U_i, \ldots, U_z, \text{ message } x \in \{0,1\}^t, \text{ and circuit } C, \]
\[ \text{can compute } U_c \text{ using homomorphic evaluation procedure described above} \]
\[ \Rightarrow \text{these procedures then satisfy the following property:} \]
\[ V_c = AU_c + C(x) \cdot G \]
\[ \|U_c\| \leq p \cdot m_{\text{G}} \]
\[ \Rightarrow \text{correspondingly, just need to set } \alpha > p \cdot m_{\text{G}} \]
\[ \Rightarrow \text{for all } \]
To verify, compute $V_c$ from $V_i$, $V_e$, and check if $V_c$, $A_{\text{lety}}$, and $H_{\text{G mold}}$ computed from signature and public parameters.