Previous lectures: SIS $\Rightarrow$ OWFs, CRHFs (symmetric cryptography) 
LWE $\Rightarrow$ PKE, key exchange (public-key cryptography) 

But we know how to do all of this before from number-theory (e.g., DHT, RSA, etc.)

Question: Do lattices give us additional power that we did not have before?

This lecture: Fully homomorphic encryption (FHE)

"Can we compute on encrypted data?"

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

Ex: ElGamal encryption:
\[
\begin{align*}
\text{pk}: (g, h = g^a) \\
\text{sk}: s
\end{align*}
\]
\[
\text{Enc}(pk, x_1) : (g^x_1, h^{x_1} g^s) \\
\text{Enc}(pk, x_2) : (g^x_2, h^{x_2} g^s) \Rightarrow (g^{x_1 + x_2}, h^{x_1 + x_2} g^s) \quad [\text{ElGamal is additively homomorphic}]
\]

Ex: Ring encryption:
\[
\begin{align*}
\text{pk}: (A, b = g \cdot A + c') \\
\text{sk}: s
\end{align*}
\]
\[
\text{Enc}(pk, x) : (A^x, b^x + x \cdot s + \frac{A^s}{A^x}) \\
\text{Enc}(pk, x_2) : (A^{x_2}, b^{x_2} + x_2 \cdot s + \frac{A^s}{A^{x_2}}) \Rightarrow (A^{x + x_2}, b^{x + x_2} + (x + x_2) \cdot s + \frac{A^s}{A^{x + x_2}})
\]

Note: in this lecture, we will write the LWE assumption as
\[
(A, g^A + e^A) \approx (A, w)
\]
\[
A \in \mathbb{Z}_q^n, s \in \mathbb{Z}_q^n, w \in \mathbb{Z}_q^n, e \in \mathbb{Z}_q
\]
this is the same assumption as in previous lectures, just transposed (often, this is a more convenient form for modern lattice constructions)

In both of these cases, we can evaluate single operation on ciphertexts (e.g., addition or multiplication) Can we support both addition and multiplication?

\[
\Rightarrow \text{Fully homomorphic encryption: encryption scheme that supports both addition and multiplication on ciphertexts (thus suffices for arbitrary computation)}
\]

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

\[
\Rightarrow \text{revolutionized lattice-based cryptography!}
\]

General blueprint: 1. Build somewhat homomorphic encryption (SWHE) — encryption scheme that supports bounded number of homomorphic operations

2. Bootstrapping SWHE to FHE (essentially a way to “refresh” ciphertext)

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

\[
\Rightarrow \text{In particular, will present Gentry-Sahai-Works (GSW) construction (conceptually simplest scheme, though not the most concretely efficient)}
\]

"3rd generation of FHE!"
Starting point: Regev encryption

KeyGen(ξ): \[ \mathbf{A} \leftarrow \mathbb{Z}_q^{m \times m}, \quad \mathbf{s} \leftarrow \mathbb{Z}_q^m, \quad e \leftarrow \mathbb{Z}_q^m \]

\[ s = \left[ \begin{array}{c} \mathbf{s} \\ 1 \end{array} \right] \in \mathbb{Z}_q^{m+1} \]

Output \( \mathsf{pk} = \mathbf{A} \) and \( \mathsf{sk} = \mathbf{s} \)

Encrypt(\( \mathsf{pk}, \mathbf{x} \)): Write \( \mathsf{pk} = \mathbf{A} \in \mathbb{Z}_q^{m \times m} \) and sample \( R \in \mathbb{Z}_q^{m \times m} \)

\[ C = \mathbf{A} \mathbf{R} + \mathbf{x} \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \quad \text{I_{(m+1)\times (m+1)} = } \left( \begin{array}{cc} I_{mm} & 0 \\ 0 & 1 \end{array} \right) \]

Decrypt(\( \mathsf{sk}, C \)): Write \( \mathsf{sk} = \mathbf{s} \). Compute \( s^T C \) and output 0 if \(| (s^T C)_m | < \frac{q}{2} \) and 1 if \(| (s^T C)_m | > \frac{q}{2} \)

\[ s^T C = s^T A R + \mathbf{x} \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T I_{(m+1)\times (m+1)} \]

Correctness: \( s^T C = s^T A R + \mathbf{x} \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T I_{(m+1)\times (m+1)} \)

Observe: the vector \( s \) (i.e., the secret key) is an approximate left eigenvector of the matrix \( C \) (i.e., the ciphertext) with associated eigenvalue \( \mathbf{x} \cdot \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \)

Security: Same as proof for Ring encryption (but hybrids: LWE, then LIL)

Observe: We can pad \( A \) with rows of all-zeros so it is a square matrix (over \( \mathbb{Z}_q^{m \times m} \)) and pad \( s \) accordingly as well.

For the ciphertext, we just embed the message in the first \((m+1)\) components.

Thus, correctness and security follow as before (value has not changed), and the message is simply the "noisy" eigenvector associated with \( s \) (the "noisy" ciphertext).

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

- Suppose \( \lambda_1 \) is a (left) eigenvalue of \( C_1 \) with associated eigenvector \( s_1 \)
- Suppose \( \lambda_2 \) is a (left) eigenvalue of \( C_2 \) with associated eigenvector \( s_2 \)

Then:

\[ s^T (C_1 + C_2) = s^T C_1 + s^T C_2 = \lambda_1 s_1^T + \lambda_2 s_2^T = (\lambda_1 + \lambda_2) s^T \]

and

\[ s^T C_2 = \lambda_1 s_1^T s_2^T \]

Works fine! For homomorphic operations:

Does the above work with approximate eigenvectors (with the padded matrices)? Unfortunately, not... Need new tricks!

Correctness: \( s^T C = \mathbf{x} \cdot \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T + e^T R \)

Addition: \( s^T (C_1 + C_2) = \mathbf{x} \cdot \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T + e^T R_1 + \mathbf{x} \cdot \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T + e^T R_2 \)

\[ = (\mathbf{x} \cdot \mathbf{x}_1) \cdot \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T + e^T (R_1 + R_2) \]

Works if \( R_1 + R_2 \) is small! (As long as \( B \ll q \), this will be OK.)

Multiplication: \( s^T C_1 C_2 = (\mathbf{x} \cdot \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T + e^T R_1) C_2 = \mathbf{x} \cdot \left[ \begin{array}{c} 1 \\ 0 \end{array} \right] \cdot s^T C_2 + e^T R_1 C_2 \)

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

This is large, since \( C_2 \) is not small!

So Correctness fails for multiplication!
The gadget matrix: A matrix with a public trapdoor (can also be viewed as a "powers-of-two" matrix)

\[ G = \begin{pmatrix} 1 & 2 & 4 & \cdots & 2^m \\ 0 & 1 & 2 & \cdots & 2^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}, \quad G \in \mathbb{Z}_2^{n \times m} \]

For notational simplicity, we will write \( n = n \lceil \log q \rceil = \Theta(n \log q) \).

a more compact way to write this is

\[ G = (1 2 4 \cdots 2^m)^\otimes \mathbf{1}_n \]

In fact, \( \|G\alpha\|_\infty = 1 \)

The magic of the gadget matrix: given any \( v \in \mathbb{Z}_q^n \), we can efficiently find a "short" \( u \in \mathbb{Z}_q^m \) such that \( Gu = v \)

\[ \text{namely SIS is easy for } G \]

\[ U = \begin{pmatrix} V_{i,1} \\ V_{i,m} \\ \vdots \\ V_{i,n} \\ \vdots \\ V_{n,1} \\ V_{n,m} \end{pmatrix} \] binary decomposition of \( V_i \)

\[ G = \begin{pmatrix} \sum_{i \in \mathbb{Z}_2} V_{i,1} \\ \sum_{i \in \mathbb{Z}_2} V_{i,2} \\ \vdots \\ \sum_{i \in \mathbb{Z}_2} V_{i,m} \end{pmatrix} \]

\[ U \rightarrow Gu = v \]

Moreover, \( \|u\|_\infty = 1 \)

In general, for a vector \( v \in \mathbb{Z}_q^n \), we write \( G^{-1}(v) \) to denote the vector \( u \in \mathbb{Z}_q^m \) consisting of the binary decomposition of the components of \( v \). More generally, if we have a matrix \( V \in \mathbb{Z}_q^{nm} \), we write \( G^{-1}(V) \) to denote applying the binary decomposition operator to each column of \( V \). Thus, we can formally define \( G^{-1} \) as the following mapping:

\[ G^{-1} : \mathbb{Z}_q^{nm} \rightarrow \mathbb{Z}_q^{nm} \]

The matrix \( G \in \mathbb{Z}_q^{nm} \) and the inverse mapping \( G^{-1} \) satisfy the following properties:

1. For all \( V \in \mathbb{Z}_q^{nm} \), \( G \cdot G^{-1}(V) = V \)
2. For all \( V \in \mathbb{Z}_q^{nm} \), \( \|G^{-1}(V)\|_\infty = 1 \)

Why is this useful? Recall previous issue with multiplication: multiplying two Reges ciphertexts \( C_1 \) and \( C_2 \) causes the error in \( C_1 \) to be scaled by \( C_2 \) and \( C_2 \) is not short.

Key idea: instead of multiplying by \( C_2 \) which is big, we instead multiply by \( G^{-1}(C_2) \), which is short. To recover correctness, we will use the property that \( G \cdot G^{-1}(C_2) = C_2 \)

The GSW Homomorphic Encryption Scheme:

**KeyGen(1^\lambda):**

\[ \tilde{s} \in \mathbb{Z}_q^n, \quad s = \frac{-\tilde{s}}{2} \in \mathbb{Z}_q^{n+1} \]

Output \( pk = \tilde{s} \) and \( sk = s \)

**Encrypt(pk, x):** Write \( pk = \tilde{s} \in \mathbb{Z}_q^{n+1} \) and sample \( \epsilon \sim f_{q}^{m \times m} \)

\[ C = AR + x : G \quad \text{[use the gadget matrix in place of the scaled identity]} \]

**Decrypt(sk, C):** Write \( sk = s \). Compute \( \tilde{s}C \) and output 0 if \( |(\tilde{s}C)_m| < \frac{s}{2} \) and 1 if \( |(\tilde{s}C)_m| > \frac{s}{2} \)

Last component is scaled by \( 2^m \): so correctness holds as long as \( B \leq q^2 \)
Conclusion: If we want to support circuits of multiplicative depth \( d \), we need to choose \( g = \Omega(d) \) to accommodate the multiplications. Observe that in this case, \( \log g = \Omega(d \log m) \), so the number of bits in the ciphertext scales linearly with the depth of the circuit.

Semantic security follows by same argument as Rags. Homomorphic operations feasible by structure of gadget matrix!

From SDHE 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 SDHE 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 SDHE scheme and an encryption of the SDHE decryption key under the SDHE public key.
- We now describe a ciphertext refreshing procedure:
  - For each SDHE 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 $ct_x = \text{Encrypt}(pk, x)$.

Using the SWHE, we can compute $ct_{f(x)} = \text{Encrypt}(pk, f(x))$ for any $f$ with multiplicative depth up to $d$

Given $ct_x$, we first compute

$$ct_c = \text{Encrypt}(pk, ct_x) \quad [\text{strictly speaking, encrypt bit by bit}]$$

This is a fresh ciphertext, so we can perform operations of depth up to $d$ on $ct_c$. Since the public key includes a copy of the decryption key ($ct_k$), we can homomorphically evaluate the decryption function:

$$
\begin{align*}
ct_{ct} &= \text{Encrypt}(pk, ct_c) \\
ct_{sk} &= \text{Encrypt}(pk, sk)
\end{align*}
$$

This is a new encryption of $m$, and we can continue performing homomorphic operations on $m$ (of depth $d - d'$)

Bootstrapping is a general technique that converts any SWHE that can evaluate its own decryption function (plus a little more) into an FHE scheme. Transformation requires additional circular security assumption (namely, that it is OK to publish an encryption of the scheme’s own public key. [The GSW scheme supports bootstrapping — decryption is a threshold inner product; choose parameters carefully])

Open problem: Build FHE from LWE (or another standard assumption) without the circular security assumption.