FULLY HOMOMORPHIC ENCRYPTION: CURRENT STATE OF THE ART

Craig Gentry, IBM Research
"I want 1) the cloud to process my data 2) even though it is encrypted.

Run

\[ \text{Eval}[ f, \text{Enc}_k(x) ] = \text{Enc}_k[f(x)] \]

The special sauce! For security parameter \( k \), Eval’s running should be \( \text{Time}(f) \cdot \text{poly}(k) \)

Delegation: Should cost less for Alice to encrypt \( x \) and decrypt \( f(x) \) than to compute \( f(x) \) herself.
Homomorphic Encryption

“Somewhat” means it works for some functions $f$

Somewhat Homomorphic Encryption (SWHE):

$\text{Enc}[x] \xrightarrow{\text{Eval}} \text{Eval} \xrightarrow{\text{Enc}[f(x)]} f$
Homomorphic Encryption

“Fully” means it works for all functions $f$.

Fully Homomorphic Encryption (FHE) [RAD78, Gen09]:

$\text{Enc}[x]$ → $\text{Eval}$ → $\text{Enc}[f(x)]$
Fully Homomorphic Encryption: Current State of the Art

- **The Good**: We have FHE with low asymptotic overhead
  - For security parameter $k$ (i.e., best attack takes time $2^k$),
  - $\text{Time}(\text{Eval}(f))/\text{Time}(f) = \text{polylog}(k)$. **Only polylog overhead!**

- **The Bad**: FHE is still very **slow in practice**.

- **The Ugly**:
  - We implemented the best current SWHE scheme.
  - For $f =$ the AES block cipher, $\text{Time}(\text{Eval}(f)) \approx 1$ day.
    - $\text{Time}(\text{Eval}(f)) \approx 37$ minutes “amortized” (for multiple AES blocks).
    - Compare: Yao garbled circuit for AES evaluated in $< 1$ second.
  - Our implementation of the best asymptotic FHE scheme is ongoing...
Overview of This Talk

- **Current state of the art:**
  - Background on somewhat homomorphic encryption (SWHE)
  - Managing the “noise” inside ciphertexts
  - “Packing” many plaintexts into each ciphertext
  - Homomorphic evaluation of the AES function with SWHE
  - Using “bootstrapping” to evaluate AES with FHE, and why we think it may speed up the evaluation of AES

- **Next steps**
Somewhat Homomorphic Encryption

with an overview of a couple of SWHE schemes...
Why Do We Care about Somewhat HE?

- **Performance**
  - For many “simple” functions, the overhead of SWHE is much less than overhead of FHE.
    - Time(Eval(f))/Time(f) is small in practice for “simple” f.
  - Rule of Thumb: If your function f can be expressed as a low-degree polynomial, SWHE might be sufficient.

- **Stepping-stone to FHE**
  - Most FHE schemes are built “on top of” a SWHE scheme with special properties.
How To Construct a SWHE Scheme?

**Most Natural Approach**

- Ciphertexts live in a “ring”.
- ADDing ciphertexts implicitly ADDs underlying plaintexts.
- Same for MULT.

**Examples of rings:** integers $\mathbb{Z}$, polynomials $\mathbb{Z}[x,y,\ldots]$, …

**“Unnatural” homomorphic encryption schemes:**
- RSA and Elgamal: Multiplying ciphertexts does multiply the plaintexts, but this is not true for addition.
- Paillier: Multiplying ciphertexts implicitly adds the plaintexts.
“Polly Cracker”: An Early Attempt at SWHE
[Fellows-Koblitz ‘93]

Main Idea
Encryptions of 0 evaluate to 0 at the secret key.

- **KeyGen**: Secret = some point \( s = (s_1, \ldots, s_n) \in \mathbb{Z}_q^n \).
  Public key: Polynomials \( \{a_i(x_1, \ldots, x_n)\} \) s.t. \( a_i(s) = 0 \mod q \).

- **Encrypt**: From \( \{a_i\} \), generate a *random* polynomial \( b(x) \) such that \( b(s) = 0 \mod q \). For \( m \) in \( \{0, 1\} \), ciphertext is:
  \[
c(x) = m + b(x) \mod q.
  \]

- **Decrypt**: Evaluate ciphertext at secret: \( c(s) = m \mod q \).

- **ADD and MULT**: Output sum or product of ciphertexts.
Polly Cracker Cryptanalysis [see AFFP11]

- An Attack:
  - Collect lots of encryptions \( \{c_i\} \) of 0.
  - If the challenge ciphertext also encrypts 0, it will likely be in linear span of the given encryptions of 0.
    - Use Gaussian elimination (linear algebra).
**Main Idea**

Encryptions of 0 evaluate to something small and even (smeven) at the secret key.

- **KeyGen**: Secret = some point $s = (s_1, \ldots, s_n) \in \mathbb{Z}_q^n$. $\gcd(q, 2) = 1$.
  Public key: Polynomials $\{a_i(x_1, \ldots, x_n)\}$ s.t. $a_i(s) = 2e_i \mod q$, $|e_i| \ll q$.

- **Encrypt**: From $\{a_i\}$, generate a *random* polynomial $b(x)$ such that $b(s) = \text{smeven} \mod q$. For $m$ in $\{0,1\}$, ciphertext is:

  $$c(x) = m + b(x) \mod q.$$ 

- **Decrypt**: Evaluate ciphertext at secret: $c(s) = m + \text{smeven} \mod q$. Then, reduce mod 2 to get $m$.

- **ADD and MULT**: Output sum or product of ciphertexts.
**Noisy Polly Cracker: A Framework for Most Current SWHE Schemes**

**Main Idea**
Encryptions of 0 evaluate to something small and even (smeven) at the secret key.

- **KeyGen**: Secret = some point $s = (s_1, \ldots, s_n) \in \mathbb{Z}^n$, $\gcd(q, 2) = 1$.
  
  Public key: Polynomials $\{a_i(x_1, \ldots, x_n)\}$ s.t. $a_i(s) \neq 0$ for all i. Generate $\{a_i\}$, generate a random polynomial $b(x)$ such that $b(s) \equiv 2e_i \mod q$. For $m$ in $\{0, 1\}$, ciphertext is:
  $$c(x) = m + b(x) \mod q.$$  
  We call $[c(s) \mod q]$ the “noise” of the ciphertext.

- **Decrypt**: Evaluate ciphertext at secret: $c(s) = m + $smeven$ \mod q$. Then, reduce mod 2 to get $m$.

- **ADD and MULT**: Output sum or product of ciphertexts.
Noisy Ciphertexts

- Each ciphertext has some noise that hides the message.
- Think: “hidden” error correcting codes…
- If error is small, Alice can use knowledge of “hidden” code to remove the noise.
- If noise is large, decryption becomes hopeless even for Alice.
SWHE with *Integers* [vDGHV10]

**Main Idea**
Encryptions of 0 are something small and even (smeven) modulo a secret integer.

- **KeyGen**: Secret key = large odd integer \(n\).
  
  Public key: Large integers \(\{a_1, \ldots, a_t\}\) such that \([a_i \mod n] = 2e_i\), where \(|e_i| \ll n\). (\(a_i\)'s are encryptions of 0.)

- **Encrypt**: From \(\{a_i\}\), generate *random* \(b\) with \(b = \text{smeven} \mod n\).
  (Roughly, \(b = \text{random-subset-sum}\{a_i\}\).) For message \(m\) in \(\{0, 1\}\), set \(c = m + b\).

- **Decrypt**: Evaluate ciphertext at secret: Compute \([c \mod n] = m + \text{smeven}\). Then, reduce mod 2 to get \(m\).

- **ADD and MULT**: Output sum or product of ciphertexts.
Security of SWHE with Integers

- **Reduction:**
  - If “approximate gcd” problem is hard, then the scheme is semantically secure.

- **Approximate GCD Problem:**
  - Given many $a_i = e_i + q_i \cdot n$ (approx multiples of $n$), output $n$.
  - Example params: $e_i \sim 2^{\Omega(k)}$, $n \sim 2^{\Omega(k^2)}$, $q_i \sim 2^{\Omega(k^5)}$, where $k$ is the security parameter
    - Best known attacks for these params (lattices) require $2^k$ time
    - But the params are huge: $\Omega(k^5)$ bits for each integer $a_i$!!!
    - Impractical! But Coron, Mandal, Naccache, and Tibouchi have recently improved the performance significantly.
Encrypting More than Bits

- So far, our plaintext space is \( \{0,1\} \): we encrypt bits.
- Can we have a bigger plaintext space?
  - Say, \( m \in \{0, \ldots, 14\} \): a “mod-15” plaintext space?
  - Sure…
SWHE with Integers (Mod-2)

Main Idea

Encryptions of 0 are something small and even (smeven) modulo a secret integer.

- **KeyGen**: Secret key = large odd integer \( n \).
  Public key: Large integers \( \{a_1, \ldots, a_t\} \) such that \( [a_i \mod n] = 2e_i \), where \( |e_i| \ll n \). (\( a_i \)'s are encryptions of 0.)

- **Encrypt**: From \( \{a_i\} \), generate random \( b \) with \( b = \text{smeven \mod n} \).
  (Roughly, \( b = \text{random-subset-sum}\{a_i\}\).) For message \( m \) in \( \{0,1\} \), set \( c = m + b \).

- **Decrypt**: Evaluate ciphertext at secret: Compute \( [c \mod n] = m + \text{smeven} \). Then, reduce mod 2 to get \( m \).

- **ADD and MULT**: Output sum or product of ciphertexts.
**Main Idea**

Encryptions of 0 are something small and even (smeeven) modulo a secret integer.

- **KeyGen:** Secret key = large integer $n$ with $\gcd(n,15)=1$.

  Public key: Large integers $\{a_1,\ldots,a_t\}$ such that $[a_i \mod n] = 15e_i$, where $|e_i| \ll n$. ($a_i$'s are encryptions of 0.)

- **Encrypt:** From $\{a_i\}$, generate random $b$ with $b = \text{smifteen} \mod n$.

  (Roughly, $b =$ random-subset-sum$\{a_i\}$.) For message $m$ in $\{0,\ldots,14\}$, set $c = m + b$.

- **Decrypt:** Evaluate ciphertext at secret: Compute $[c \mod n] = m + \text{smifteen}$. Then, reduce mod 15 to get $m$.

- **ADD and MULT:** Output sum or product of ciphertexts.
A More Complicated SWHE Scheme…

- Sadly, we proceed to a more complicated SWHE scheme…

- Instead of $\mathbb{Z}$ (the integers), we will use a more complicated ring: the polynomial ring $R = \mathbb{Z}[x]/\Phi_N(x)$

- Why?
  - Smaller parameters (better efficiency).
  - Pack many plaintexts into each ciphertext.
Cyclotomic Ring $R = \mathbb{Z}[x]/\Phi_N(x)$

- $\Phi_N(x)$ is the $N$-th cyclotomic polynomial
  - Largest irreducible divisor of $x^{N-1}$.
  - The degree of $\Phi_N(x)$ is $n = \varphi(N) < N$.
    - $\varphi$ is Euler totient function. E.g., for $N = pq$, $\varphi(N) = (p-1)(q-1)$.
    - But we use much smaller $N$ than RSA. Example: $N = 68561$, $\varphi(N) = 62208$.
  - We call $n (=\varphi(N))$ the “ring dimension”.

- Properties of $R$
  - $R$ consists of polynomials of degree $n-1$.
  - $R$ has an additive identity (0) and multiplicative identity (1).
  - Addition: From $a(x), b(x) \in R$, compute $a(x)+b(x)$ in usual way.
  - Multiplication: From $a(x), b(x) \in R$, compute $c(x)=a(x) \cdot b(x)$, and then reduce $c(x)$ modulo $\Phi_N(x)$.

- $R_q = \mathbb{Z}_q[x]/\Phi_N(x)$ – i.e., elements of $R$ with coefficients reduced mod $q$. 
Main Idea

Encryptions of 0 have a small and even (smeven) dot product with the secret key.

- **KeyGen:** Secret = some point $s$ s.t. \( \gcd(q, 2) = 1 \).
  
  Public key: **Linear** polynomials $a_i(y) = a_{i0} + a_{i1}y \in \mathbb{R}[y]$ s.t. $a_i(s) = 2e_i \mod q$, where $e_i$ is an element of $\mathbb{R}$ with $|e_i| \ll q$.

- **Encrypt:** From $\{a_i\}$, generate a **random linear** polynomial $b(y)$ such that $b(s) = \text{smeven} \mod q$ (via subset sum). For $m$ in $\mathbb{R}_2$, ciphertext is:
  
  $$c(y) = m + b(y) \mod q$$  (a linear polynomial).

- **Decrypt:** Evaluate ciphertext at secret key:
  
  $$c(y) = c_0 + c_1s = m + \text{smeven} \mod q.$$  Reduce mod 2.

- **ADD and MULT:** Output sum or product of ciphertexts. Relinearize.
Security of SWHE over Cyclotomic Rings

- The Ring Learning-with-Errors (RLWE) Problem
  - Let $B \ll q$. Choose $s$ randomly from $R_q$. Given many (linear) polynomials $a_i(y) = a_{i0} + a_{i1} \cdot y \in R[y]$ such that $a_i(s) = e_i$ (an $R_q$-element whose coefficients are smaller than $B$) output $s$.

- How hard is the RLWE Problem?
  - [LPR10]: As hard as solving the $(q/B)$-approximate shortest vector problem (SVP) on ideal lattices in $R$.
  - As $q/B$ increases, RLWE becomes easier (unless $n$ grows).
    - For $B \sim \text{poly}(k)$, best known attacks for require $2^k$ time when $n$ (the ring dimension) $\sim k \log(q/B)$. 
Parameters and Performance

- If \( q = \text{poly}(k) \), then \( n \sim O(k) \).
  - Ciphertexts consist of 2 elements of \( \mathbb{R}_q \): only \( 2n \log q = \tilde{O}(k) \) bits.
  - ADD and MULT: \( \tilde{O}(k) \) computation. (Use FFT for MULT.)

- If \( q = 2^k \), then \( n \sim O(k^2) \). (\( n \) is large so to make RLWE hard.)
  - Ciphertext size and homomorphic ops take \( \tilde{O}(k^3) \) computation.

- Why would you consider making \( q \) so big?
  - Homomorphic ops (especially MULTs) make ciphertext noise grow.
    - Noise grows exponentially with degree of function being evaluated.
  - Decryption error occurs if noise becomes bigger than \( q \).
Managing Ciphertext Noise
Better Noise Management?

- Our fantasy:
  - Noise doesn’t grow exponentially with degree.
  - There is some *simple* trick to reduce noise after MULTs.
  - We get better noise management, hence shorter ciphertexts and SWHE for bounded depth.
Better Noise Management?

- **Crazy Idea [BV11b, BGV12]:**
  - Suppose $c$ encrypts $m$ — that is, $m = [[c(s)]_q]_2$.
  - Let’s pick $p < q$ and set $c^*(x) = (p/q) \cdot c(x)$, rounded.
  - Maybe it is true that:
    - $c^*(x)$ encrypts $m$: $m = [[c^*(s)]_p]_2$ (new inner modulus).
    - $|c^*(s)]_p| \approx (p/q) \cdot |c(s)]_q|$ (noise is smaller).
  - This really shouldn’t work…

- **Bottom line:** Crazy idea (**modulus switching**) basically works, and our fantasy comes true!
How Does Modulus Switching Help? [BGV12]

To evaluate a circuit of depth $L$...

- Start with a large modulus $q_L$ and noise of size $\eta \ll q_L$.
- After first MULT, noise grows to size $\eta^2$.
- Switch the modulus to $q_{L-1} \approx q_L/\eta$.
  - Noise reduced to $\eta^2/\eta \approx \eta$.
- After next MULT, noise again grows to $\eta^2$. Switch to $q_{L-2} \approx q_{L-1}/\eta$ to reduce the noise to $\eta$.
- Keep switching moduli after each layer.
  - Setting $q_{i-1} \approx q_i/\eta$. (“Ladder” of decreasing moduli.)
  - Until the last modulus just barely satisfies $q_0 > \eta$. 

How Does Modulus Switching Help? [BGV12]

- **Example:** $q_8 \approx \eta^9$ with/without modulus reduction.

<table>
<thead>
<tr>
<th>Level</th>
<th>Degree</th>
<th>With</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresh ciphertexts</td>
<td></td>
<td>$\eta$</td>
<td>$q_8 = \eta^9$</td>
</tr>
<tr>
<td>Level-1, Degree=2</td>
<td></td>
<td>$\eta$</td>
<td>$q_7 = \eta^8$</td>
</tr>
<tr>
<td>Level-2, Degree=4</td>
<td></td>
<td>$\eta$</td>
<td>$q_6 = \eta^7$</td>
</tr>
<tr>
<td>Level-3, Degree=8</td>
<td></td>
<td>$\eta$</td>
<td>$q_5 = \eta^6$</td>
</tr>
<tr>
<td>Level-4, Degree=16</td>
<td></td>
<td>$\eta$</td>
<td>$q_4 = \eta^5$</td>
</tr>
<tr>
<td>Level-5, Degree=32</td>
<td></td>
<td>$\eta$</td>
<td>$q_3 = \eta^4$</td>
</tr>
<tr>
<td>Level-6, Degree=64</td>
<td></td>
<td>$\eta$</td>
<td>$q_2 = \eta^3$</td>
</tr>
<tr>
<td>Level-7, Degree=128</td>
<td></td>
<td>$\eta$</td>
<td>$q_1 = \eta^2$</td>
</tr>
<tr>
<td>Level-8, Degree=256</td>
<td></td>
<td>$\eta$</td>
<td>$q_0 = \eta$</td>
</tr>
</tbody>
</table>
How Does Modulus Switching Help? [BGV12]

- Example: $q_8 \approx \eta^9$ with/without modulus reduction.

<table>
<thead>
<tr>
<th></th>
<th>With</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Noise</td>
<td>Modulus</td>
</tr>
<tr>
<td>Fresh ciphertexts</td>
<td>$\eta$</td>
<td>$q_8 = \eta^9$</td>
</tr>
<tr>
<td>Level-1, Degree=2</td>
<td>$\eta$</td>
<td>$q_7 = \eta^8$</td>
</tr>
<tr>
<td>Level-2, Degree=4</td>
<td>$\eta$</td>
<td>$q_6 = \eta^7$</td>
</tr>
<tr>
<td>Level-3, Degree=8</td>
<td>$\eta$</td>
<td>$q_5 = \eta^6$</td>
</tr>
<tr>
<td>Level-4, Degree=16</td>
<td>$\eta$</td>
<td>$q_4 = \eta^5$</td>
</tr>
<tr>
<td>Level-5, Degree=32</td>
<td>$\eta$</td>
<td>$q_3 = \eta^4$</td>
</tr>
<tr>
<td>Level-6, Degree=64</td>
<td>$\eta$</td>
<td>$q_2 = \eta^3$</td>
</tr>
<tr>
<td>Level-7, Degree=128</td>
<td>$\eta$</td>
<td>$q_1 = \eta^2$</td>
</tr>
<tr>
<td>Level-8, Degree=256</td>
<td>$\eta$</td>
<td>$q_0 = \eta$</td>
</tr>
</tbody>
</table>

Decryption error
Where Are We? (The Good News)

- We have efficient SWHE for circuits of polynomially bounded depth.
  - To evaluate circuit of depth $L$:
    - Largest modulus is $q_L \approx q_0^L \approx \eta^L$.
    - Largest ciphertext is $O(k \cdot \text{poly}(L))$ bits for security param $k$.
      - “Efficient” in the sense of being polynomial size.
Where Are We? (The Bad News)

- But the overhead \(\text{Time}(\text{Eval}(f))/\text{Time}(f)\) seems large
  - We are not using our plaintext space optimally.
    - Plaintext \(m\) is in \(\mathbb{R}^2\). \(m\) is \(O(k)\) bits “in principle”.
    - But can we actually use more than 1 bit of this “weird” plaintext space?
  - The overhead depends on \(L\), the depth of the circuit for \(f\).
    - Ciphertext is \(O(k \cdot \text{poly}(L))\) bits
    - Can we remove the dependence on \(L\)?
Packing Plaintexts into Ciphertexts
Weird Plaintext Spaces

- SWHE with integers, using a mod-15 message space
  - Recall: $c$ is decrypted as $[c \mod n] \mod 15$.

- Message space is $\mathbb{Z}_{15}$
  - $\mathbb{Z}_{15} = \mathbb{Z}_3 \times \mathbb{Z}_5$. Chinese Remainder Theorem (CRT).
  - From one “big” plaintext space we get 2 independent “small” plaintext spaces. We call them 2 “plaintext slots”.

- Suppose two ciphertexts $c$ and $c'$ have $(r_3, r_5)$ and $(r_3', r_5')$ in their respective mod-3 and mod-5 “plaintext slots”
  - $c_{\text{ADD}} = \text{ADD}(c, c')$ has $(r_3 + r_3', r_5 + r_5')$ in its slots.
  - $c_{\text{MULT}} = \text{MULT}(c, c')$ has $(r_3 \cdot r_3', r_5 \cdot r_5')$ in its slots.
  - Homomorphic ops act component-wise, in parallel, on slots.
Our Weird Cyclotomic Plaintext Space

- SWHE based on RLWE [BV11, BGV12]
  - Plaintext space is $R_2 = \mathbb{Z}_2[x]/\Phi_N(x)$.
    - $m(x)$ is a polynomial in $R = \mathbb{Z}[x]/\Phi_N(x)$, with coefficients in $\{0,1\}$.
    - $m$ has $n$ bits, where $n$ is the degree of $\Phi_N(x)$.
  - Ciphertext $c(y) = c_0 + c_1 y$ decrypted as $(c(s) \mod q) \mod 2$.

- Can we get many “plaintext slots” out of $R_2$?
  - Sure…
Our Weird Cyclotomic Plaintext Space

Main Point

The plaintext space $R_2 = \mathbb{Z}_2[x]/\Phi_N(x)$ has amazing properties! Much better than a mod-15 plaintext space!

- Via CRT, $R_2$ decomposes into about $N/\log(N)$ plaintext slots of about $\log(N)$ bits apiece (for well-chosen $N$).
  - ADD and MULT work in parallel across the slots.

- Via ring automorphisms, encrypted data can be moved between slots.
  - We have ADD, MULT, and PERMUTE.

- Can evaluate boolean circuits with ciphertexts “packed”.
  - Reduces overhead.
SIMD (Single Instruction Multiple Data): Working on Data Arrays

Array of length $n$

2 1 9 5 0 7 3 6 ... 1 2
8 2 0 9 3 8 0 1 ... 4 4
10 3 9 14 3 15 3 7 ... 5 6

$n$-ADD
SIMD (Single Instruction Multiple Data): Working on Data Arrays

Array of length n

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>9</th>
<th>5</th>
<th>0</th>
<th>7</th>
<th>3</th>
<th>6</th>
<th>...</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

| 16 | 2 | 0 | 45 | 0 | 56 | 0 | 6 | ... | 4 | 8 |

n-MULT
SIMD (Single Instruction Multiple Data): Working on Data Arrays

Array of length $n$

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>9</th>
<th>5</th>
<th>0</th>
<th>7</th>
<th>3</th>
<th>6</th>
<th>...</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td>...</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td>...</td>
<td>%</td>
<td>%</td>
</tr>
</tbody>
</table>

- Great for computing same function $F$ on $n$ different input strings.
- We can do SIMD homomorphically.
With $n$-ADD, $n$-MULT, and $n$-PERMUTE, we can evaluate arbitrary circuits without “unpacking” ciphertexts.
A Taste of Cyclotomic Rings

- Choose \( N \) so that \( \Phi_N(x) \) factors mod 2 into \( t \) factors.
  - \( \Phi_N(x) = \prod f_i(x) \mod 2 \). Degrees of \( f_1, \ldots, f_t \) are \( d = \phi(N)/t \).

- Chinese Remainder Theorem (CRT) – polynomial version
  - \( \mathbb{Z}_2[x]/\Phi_N(x) = \mathbb{Z}_2[x]/f_1(x) \times \ldots \times \mathbb{Z}_2[x]/f_t(x) \)

- If ciphertexts \( c \) and \( c' \) have \( (r_1(x),\ldots,r_t(x)) \) and \( (r_1'(x),\ldots,r_t'(x)) \) in their respective plaintext slots
  - \( c_{\text{ADD}} = \text{ADD}(c,c') \) has \( (r_1(x)+r_1'(x),\ldots,r_t(x)+r_t'(x)) \).
  - \( c_{\text{MULT}} = \text{MULT}(c,c') \) has \( (r_1(x)\cdot r_1'(x) \mod f_1(x), \ldots, r_t(x) \cdot r_t'(x) \mod f_t(x)) \).
  - Homomorphic ops act component-wise, in parallel, on slots.
A Taste of Cyclotomic Rings II

- Ring automorphisms
  - \( m(x) + 2e(x) = c_0(x) + c_1(x)s(x) \mod (q, x^{N-1}) \)
  - \( m(x^k) + 2e(x^k) = c_0(x^k) + c_1(x^k)s(x^k) \mod (q, x^{kN-1}) \)
  - \( m(x^k) + 2e(x^k) = c_0(x^k) + c_1(x^k)s(x^k) \mod (q, x^{N-1}) \).
    - That is, we get ciphertext \((c_0(x^k), c_1(x^k))\) encrypts \(m(x^k)\) under key \(s(x^k)\), “for free”
  - Use “key-switching” to return to key \(s(x)\).

- How is the new message \(m(x^k)\) related to the original message \(m(x)\)?
  - Suppose \(k\) is a power of 2
    - \(m(x)^{2^i} = m(x^{2^i}) \mod 2\). (Frobenius automorphism)
    - We can exponentiate, in parallel, the plaintext slots by a power of 2 “for free”.
  - For certain other values of \(k\)
    - Map \((x \rightarrow x^k)\): \(m(\alpha_1^k), \ldots, m(\alpha_t^k)\) is a permutation of \(m(\alpha_1), \ldots, m(\alpha_t)\).
    - We can “rotate” the slots.
      - More tricks are needed for general permutations of the slots.
What Is Our Overhead Now?

- A ciphertext is still $O(k \cdot \text{poly}(L))$ bits
  - Recall: $k$ is the security parameter, and $L$ is the number of levels in the circuit we are evaluating

- But ciphertext is “packed” with about $k/\log(k)$ slots

- The overhead is now only $\text{polylog}(k) \cdot \text{poly}(L)$
Homomorphic Evaluation of AES
AES (Advanced Encryption Standard)

- AES-128 block cipher: 10 applications of a round function.
- The round function operates on a $4 \times 4$ matrix of bytes, where each byte may be viewed as an element of $F_{2^8}$.
- The round function consists of 4 basic sub-functions:
  - AddKey: XOR current state with 16 byte key.
  - SubBytes:
    - First, an inversion in $F_{2^8}$.
    - Next, a fixed $F_2$-linear map on the bits of the element.
  - ShiftRows: rotates the bytes within each row.
  - MixColumns: a linear mixing operation within each column.

The only non-linear step in AES.
Our Plaintext Space for AES

- We make our plaintext “slots” are isomorphic to $\mathbb{F}_{2^8}$.
  - More accurately, to $\mathbb{F}_{2^d}$ where 8 divides $d$, and $d$ is the degree of the polynomials $f_i(x)$ that divide $\Phi_N(x)$.
  - Each slot holds a byte of the AES state.
The Inversion Step of AES

- For $m \in \mathbb{F}_{2^8}$, we have that $m^{-1} = m^{254}$.
- We can homomorphically compute encryptions of $m, m^2, m^4, \ldots, m^{128}$ “for free” (Frobenius aut.)
- $m^{254} = m^{128} \times m^{64} \times m^{32} \times m^{16} \times m^{8} \times m^{4} \times m^{2}$.
  - A product of 7 things
  - Computable by a circuit with 3 levels (binary tree).

- Each AES round actually costs us about 6 levels.
  - Other steps are not completely “free”.
  - $L = 60$ for 10 round AES.
Parameter Sizes

- “Ladder” of odd moduli $q_0 < q_1 < \ldots < q_L$.
  - For AES, $L=60$
  - Example $q_i = q_0^{i+1}$, where $q_0 = \text{poly}(k)$.

- For security, ring dimension $N \approx k \log(q_L/B) = O(kL)$

- Ciphertext size $= O(N \log q)$ bits $\approx \tilde{O}(k L^2)$ bits.
## Parameter Sizes

<table>
<thead>
<tr>
<th>L (levels)</th>
<th>N</th>
<th>n = φ(N)</th>
<th>(slot size, #slots)</th>
<th>log(q_L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11441</td>
<td>10752</td>
<td>(48,224)</td>
<td>177</td>
</tr>
<tr>
<td>20</td>
<td>34323</td>
<td>21504</td>
<td>(48,448)</td>
<td>368</td>
</tr>
<tr>
<td>30</td>
<td>31609</td>
<td>31104</td>
<td>(72,432)</td>
<td>564</td>
</tr>
<tr>
<td>40</td>
<td>54485</td>
<td>40960</td>
<td>(64,640)</td>
<td>762</td>
</tr>
<tr>
<td>50</td>
<td>59527</td>
<td>51840</td>
<td>(72,720)</td>
<td>962</td>
</tr>
<tr>
<td>60</td>
<td>68561</td>
<td>62208</td>
<td>(72,864)</td>
<td>1163</td>
</tr>
<tr>
<td>70</td>
<td>82603</td>
<td>75264</td>
<td>(56,1344)</td>
<td>1366</td>
</tr>
<tr>
<td>80</td>
<td>92837</td>
<td>84672</td>
<td>(56,1512)</td>
<td>1570</td>
</tr>
</tbody>
</table>

- For L=60, ciphertext size is about \(2^n \log q = 2 \times 62208 \times 1163 \approx 14\) million bits.
Running Times

- Run a one-core machine with lots of RAM (256GB)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Levels Needed</td>
<td>60</td>
</tr>
<tr>
<td>Key Generation</td>
<td>43 minutes</td>
</tr>
<tr>
<td>Encrypt AES State</td>
<td>2 minutes</td>
</tr>
<tr>
<td>Encrypt AES Key Schedule</td>
<td>23 minutes</td>
</tr>
<tr>
<td>Evaluate AES Round 1</td>
<td>7 hours</td>
</tr>
<tr>
<td>Evaluate AES Round 9</td>
<td>2 hours</td>
</tr>
<tr>
<td>Evaluate AES Round 10</td>
<td>28 minutes</td>
</tr>
<tr>
<td>Evaluate AES total</td>
<td>34 hours</td>
</tr>
<tr>
<td>Number of SIMD Blocks</td>
<td>54</td>
</tr>
<tr>
<td>Time Per Block</td>
<td>37 minutes</td>
</tr>
</tbody>
</table>
Bootstrapping
Bootstrapping: What’s the Motivation?

- So far, the overhead grows polynomially with $L$.
  - For example, ciphertext size grows with $L^2$.

- Want overhead to be independent on $L$.
  - To only depend on the security parameter $k$.

- Achievable!
  - We can make the overhead $\text{polylog}(k)$ asymptotically.
  - But the overhead is still very large in practice.
Bootstrapping: What Is It?

- So far, we can evaluate bounded depth funcs $F$: 
  $$F(x_1, x_2, \ldots, x_t)$$

- We have a noisy evaluated ciphertext $c$.
- We want to get a less noisy $c'$ that encrypts the same value, but with less noise.
- Bootstrapping refreshes ciphertexts, using the encrypted secret key.
Bootstrapping: What Is It?

- For ciphertext $c$, consider $D_c(sk) = \text{Decrypt}_{sk}(c)$
  - Suppose $D_c(\cdot)$ is a low-depth polynomial in $sk$.
- Include in the public key also $\text{Enc}_{pk}(sk)$.

Homomorphic computation applied only to the “fresh” encryption of $sk$.

New encryption of $\gamma$, with less noise.
Set $L > 1 + \text{depth needed to evaluate } D_C$.
- Decryption depth is only logarithmic in security parameter.

We now have “pure” FHE.
- Use homomorphic decryption recursively to keep noise level down.
Applying Bootstrapping to AES

- For AES, L=60 (very large)
- We think we can evaluate the decryption function using fewer levels – e.g., 10
  - If so, it may pay off to “refresh” (bootstrap) after every 1 or 2 rounds of AES evaluation.
Summary
In 3 years, we have collected a big bag of tricks
- SWHE schemes that allow ADD and MULT
- Tricks to prevent ciphertext noisiness from growing too fast
- Embedding many “plaintext slots” into each ciphertext
  - And ways of moving data around within the slots
- And our last (best?) resort: bootstrapping

We can evaluate AES homomorphically in time that is only moderately ridiculous.

For simpler functions, SWHE may actually be practical.
Next Steps
Can SWHE Be Practical? [LNV11]

- Implementation of [BV11a] SWHE scheme, limited to evaluation of functions with very low degree – e.g., 2.
- For ring dimension 2048, Mult takes 43 msec.
  - Comparable to 23 msec of [GH10]
  - They use Intel Core 2 Duo Processor at 2.1 GHz.
- Shows lattice-based SWHE can compute quadratic functions more efficiently than [BGN05] (pairing-based)

Another possibility: tailoring the SWHE scheme for the particular function being evaluated

- We did this for AES to a small extent.
Open Problem: A Fast Refresh Procedure

- Bootstrapping is one way to reduce ciphertext noise
  - Unlike modulus reduction, it can be used repeatedly

- Can we find a better reusable noise-reduction trick?
  - There is no (known) inherent reason why noise-reduction must involve homomorphic evaluation of the decryption function….
Thank You! Questions?
Homomorphic Encryption

“Fully” means it works for all functions $f$

Fully Homomorphic Encryption (FHE) [RAD78, Gen09]:

- Building FHE from SWHE [Gen09]:
  - Requires a bootstrapping Currently requires a computationally expensive step called “bootstrapping” or “recryption.”
Homomorphic Encryption

“Fully” means it works for all functions \( f \).

Fully Homomorphic Encryption (FHE) [RAD78, Gen09]:

- [Gen09]: First FHE scheme.
- Overhead of FHE: \( \text{Time}(\text{Eval}(f))/\text{Time}(f) = \text{poly}(k) \)
  - \( k \) is the security parameter (best attack takes time \( 2^k \)).
SWHE with Integers [vDGHV10]

Main Idea
Encryptions of 0 are something small and even (smeven) modulo a secret integer.

- **KeyGen**: Secret key = large odd integer n.
  Public key: Large integers \(\{a_1, \ldots, a_t\}\) such that \([a_i \mod n] = 2e_i\), where \(|e_i| \ll n\). (\(a_i\)'s are encryptions of 0.)

- **Encrypt**: From \(\{a_i\}\), generate *random* \(b\) with \(b = \text{smeven} \mod n\).
  (Roughly, \(b = \text{random-subset-sum}\{a_i\}\).) For message \(m\) in \(\{0,1\}\), set \(c = m + b\).

- **Decrypt**: Evaluate ciphertext at secret: Compute \([c \mod n] = m + \text{smeven}\). Then, reduce mod 2 to get \(m\).

- **ADD and MULT**: Output sum or product of ciphertexts.
SWHE Based on Learning-with-Errors (LWE) [Brakerski-Vaikuntanathan ‘11]

**Main Idea**

Encryptions of 0 have a small and even (smeven) dot product with the secret key.

- **KeyGen**: Secret = some point $s = (s_1, \ldots, s_n) \in \mathbb{Z}_q^n$.
  Public key: **Linear** polys $\{f_i(x_1, \ldots, x_n)\}$ s.t. $f_i(s) = 2e_i \mod q$, $|e_i| \ll q$.
  That is, $f_i(s) = f_{i0} + f_{i1} \cdot s_1 + \ldots + f_{in} \cdot s_n = \langle f_i, (1, s) \rangle = e_i$.

- **Encrypt**: From $\{f_i\}$, generate random linear polynomial $g(x)$ such that $g(s) = \text{smeven} \mod q$ (via subset sum). For $m$ in $\{0,1\}$, ciphertext is:
  $$c(x) = m + g(x) \mod q.$$

- **Decrypt**: Evaluate ciphertext at secret: $c(s) = m + \text{smeven} \mod q$.
  Then, reduce mod 2 to get $m$.

- **ADD and MULT**: Output sum or product of ciphertexts. Relinearize.
Relinearizing and “Key Switching”

- “Normal” ciphertext is a linear polynomial \( c(y) = c_0 + c_1 y \).

- **MULT** outputs a quadratic ciphertext
  \[ \text{MULT}(c(y), c'(y)) \rightarrow c''(y). \quad c''(y) = (c_0 + c_1 y)(c_0' + c_1' y) = c_1 c_1' y^2 + \ldots \]
  - Decryption works: \( m'' = m \cdot m' = (c''(s) \mod q) \mod 2 \)
  - But… It would be better if \( c''(y) \) were “normal”.

- **Relinearize:**
  - Evaluator can use \( \text{aux} \) to transform \( c''(y) \) to a linear ciphertext \( c^\dagger(y) \) that encrypts \( m'' \) under \( s \).
  - Augment public key with certain auxiliary material \( \text{aux} \).

- **Key Switching**
  - More generally, augment \( \text{pk} \) with \( \text{aux}(s_1, s_2) \) for keys \( s_1 \) and \( s_2 \).
  - Evaluator can transform ciphertext \( c(y) \) under \( s_1 \) to \( c'(y) \) under \( s_2 \).
  - Like proxy re-encryption.
Security of LWE-Based SWHE

- **Reduction:**
  - If “learning with errors (LWE)” problem is hard, then the scheme is semantically secure

- **LWE Problem:**
  - Given many linear polys $f_i(x)$ with $[f_i(s)]_q = e_i$ (small), output $s = (s_1, \ldots, s_n)$.
  - Example params: $n \sim k \log q$, $e_i \sim \text{poly}(n)$, where $k$ is the security parameter
    - Best known attacks for these params (lattices) require $2^k$ time
    - But relinearization step is expensive!: $\Omega(n^3)$. 
Basic PERMUTEs (Rotations)

Array of length n

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
<th>n-2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i+1</td>
<td>i+2</td>
<td>i+3</td>
<td>i+4</td>
<td>i+5</td>
<td>i+6</td>
<td>i+7</td>
<td>...</td>
<td>i-2</td>
<td>i-1</td>
</tr>
</tbody>
</table>

- **n-ROTATE(i)**
n-PERMUTE: How do we do it?

- The “Basic” Permutations \((b(y) = a(y^i))\):
  - Ring automorphism \(a(y) \rightarrow a(y^i)\) give us “basic permutations” like “rotations”
    - “Rotations”: \(n\)-ROTATE\((i)\), which rotates the \(n\) plaintext slots \(i\) steps clockwise, like a dial.

- Benes network
  - A Benes network consists of 2 conjoined butterfly networks
  - Use a Benes network to allow evaluate complicated permutations from the basic ones.
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)

Potential Swaps
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)

1 2 3 4 5 6 7 8
7 8 1 2 3 4 5 6
3 4 5 6 7 8 1 2

Actual Swaps
8-ROTATE(2)
8-ROTATE(-2)
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)

```
1 2 3 4 5 6 7 8
7 8 1 2 3 4 5 6
3 4 5 6 7 8 1 2
```

Actual Swaps
8-ROTATE(2)
8-ROTATE(-2)
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)

```
1 0 1 0 0 1 0 1
1 2 3 4 5 6 7 8
7 8 1 2 3 4 5 6
3 4 5 6 7 8 1 2
```
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)
n-PERMUTE(π) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)

```
10300608
00020050
34567812
```
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)

```
 1 0 3 0 0 6 0 8
```

```
0 0 0 2 0 0 5 0
```

```
0 1 0 0 1 0 0 0
```

```
3 4 5 6 7 8 1 2
```

n-MULT
n-PERMUTE(\(\pi\)) from n-ADD, n-MULT, and n-ROTATE(i). (Sketch)

- 8-SWAP(2,0110)

```
1 0 3 0 0 6 0 8
0 0 0 2 0 0 5 0
0 4 0 0 7 0 0 0
```
\( n\text{-PERMUTE}(\pi) \) from \( n\text{-ADD} \), \( n\text{-MULT} \), and \( n\text{-ROTATE}(i) \). (Sketch)

- **8-SWAP(2,0110)**

\[
\begin{array}{cccccc}
1 & 0 & 3 & 0 & 0 & 6 \\
0 & 0 & 0 & 2 & 0 & 0 \\
0 & 4 & 0 & 0 & 7 & 0 \\
1 & 4 & 3 & 2 & 7 & 6
\end{array}
\]

\( n\text{-ADD} \)

\[
\begin{array}{cccccc}
8 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 4 & 0 & 0 & 0 & 0 \\
1 & 4 & 3 & 2 & 7 & 6
\end{array}
\]