Lecture 11: Fully homomorphic encryption

Lecturer: Eran Tromer

Including presentation material by Vinod Vaikuntanathan, MIT
Fully Homomorphic Encryption
Confidentiality of static data: plain encryption
Confidentiality of data inside computation: Fully Homomorphic Encryption
Fully Homomorphic Encryption

• Goal: delegate computation on data without revealing it
• A confidentiality goal
Delegate **processing** of data without **revealing** it

**Example 1: Private search**

- **You:** Encrypt the query, send to Google  
  (Google does not know the key, cannot “see” the query)

- **Google:** Encrypted query → Encrypted results  
  (You decrypt and recover the search results)
Example 2: Private Cloud Computing

Delegate processing of data without revealing it

(Input: $x$) ($Program: P$) $\rightarrow$ $Enc(P(x))$
Fully Homomorphic Encryption

Encrypted x, Program P → Encrypted P(x)

Definition: \((\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval})\)

(as in regular public/private-key encryption)

- **Correctness of Eval**: For every input x, program P
  - If \(c = \text{Enc}(PK, x)\) and \(c' = \text{Eval}(PK, c, P)\),
  then \(\text{Dec}(SK, c') = P(x)\).

- **Compactness**: Length of \(c'\) independent of size of P

- **Security**: semantic security / indistinguishability \([GM82]\)
History of Fully Homomorphic Encryption

- **First Defined:**
  “Privacy homomorphism”
  [Rivest Adleman Dertouzos 78]
  motivation: searching encrypted data

- **Limited homomorphism:**
  - RSA & El Gamal: multiplicatively homomorphic
    multiply ciphertexts $\mapsto$ multiply plaintext
  - GM & Paillier: additively homomorphic
    plaintext in exponent
    multiply ciphertext $\mapsto$ add plaintext
  - Quadratic formulas
    [BGN 05] [GHV 10]

- **Non-compact homomorphic encryption:**
  - Based on Yao garbled circuits
  - [SYY 99] [MGH 08]: $c^*$ grows exp with degree/depth
  - [IP 07] branching programs

\[ c_1 = m_1^e \quad c_2 = m_2^e \quad c_3 = m_3^e \]
\[ c^* \equiv c_1 c_2 c_3 \equiv (m_1 m_2 m_3)^e \pmod{n} \]
Big Breakthrough: [Gentry09]

First Construction of Fully Homomorphic Encryption
using algebraic number theory & “ideal lattices”

► Full-semester course
► Today: an alternative construction [DGHV 10]
  – using just integer addition and multiplication
  – easier to understand, implement and improve

Eval: $P$, $\text{Enc}(x) \rightarrow \text{Enc}(P(x))$
Constructing fully-homomorphic encryption assuming hardness of approximate GCD
A Roadmap

1. Secret-key “Somewhat” Homomorphic Encryption
   (under the approximate GCD assumption)
   (a simple transformation)

2. Public-key “Somewhat” Homomorphic Encryption
   (under the approximate GCD assumption)
   (borrows from Gentry’s techniques)

3. Public-key FULLY Homomorphic Encryption
   (under approx GCD + sparse subset sum)
Secret-key Homomorphic Encryption

1. **Secret key**: a large odd number $p$ (sec. param = $n$)

2. **To Encrypt a bit $b$**:
   - pick a random “large” multiple of $p$, say $q \cdot p$ (q ~ $n^5$ bits)
   - pick a random “small” even number $2 \cdot r$ (r ~ $n$ bits)
   - Ciphertext $c = q \cdot p + 2 \cdot r + b$

3. **To Decrypt a ciphertext $c$**:
   - $c \pmod{p} = 2 \cdot r + b \pmod{p}$
   - read off the least significant bit
How to Add and Multiply Encrypted Bits:

- Add/Mult two near-multiples of $p$ gives a near-multiple of $p$.

\[-c_1 = q_1 \cdot p + (2 \cdot r_1 + b_1), \quad c_2 = q_2 \cdot p + (2 \cdot r_2 + b_2)\]

\[-c_1 + c_2 = p \cdot (q_1 + q_2) + 2 \cdot (r_1 + r_2) + (b_1 + b_2) \quad \ll p\]

\[\text{LSB} = b_1 \text{ XOR } b_2\]

\[-c_1c_2 = p \cdot (c_2q_1 + c_1q_2 - q_1q_2) + 2 \cdot (r_1r_2 + r_1b_2 + r_2b_1) + b_1b_2 \quad \ll p\]

\[\text{LSB} = b_1 \text{ AND } b_2\]
Problems

1. **Ciphertext grows** with each operation

   - Useless for many applications (cloud computing, searching encrypted e-mail)

2. **Noise grows** with each operation

   - Consider $c = qp+2r+b \leftarrow \text{Enc}(b)$
   - $c \mod p = r' \neq 2r+b$
   - $\text{lsb}(r') \neq b$
Problems

1. **Ciphertext grows** with each operation
   - Useless for many applications (cloud computing, searching encrypted e-mail)

2. **Noise grows** with each operation
   - Can perform “limited” number of hom. operations
   - What we have: “**Somewhat Homomorphic**” Encryption
Public-key Homomorphic Encryption

1. Secret key: an $n^2$-bit odd number $p$

Public key: $[q_0p+2r_0, q_1p+2r_1, ..., q_tp+2r_t] \overset{\triangle}{=} (x_0, x_1, ..., x_t)$

- $t+1$ encryptions of 0
- Wlog, assume that $x_0$ is the largest of them

3. To Decrypt a ciphertext $c$:

- $c \pmod{p} = 2\cdot r+b \pmod{p} = 2\cdot r+b$
- read off the least significant bit

4. Eval (as before)
Public-key Homomorphic Encryption

1. Secret key: an $n^2$-bit odd number $p$

   Public key: $[q_0p+2r_0, q_1p+2r_1, \ldots, q_tp+2r_t] \triangleq (x_0, x_1, \ldots, x_t)$

2. To Encrypt a bit $b$: pick random subset $S \subseteq [1\ldots t]$

   $$c = \sum_{i \in S} x_i + 2r + b \pmod{x_0}$$

3. To Decrypt a ciphertext $c$:

   $- c \pmod{p} = 2 \cdot r + b \pmod{p} = 2 \cdot r + b$

   - read off the least significant bit

4. Eval (as before)
Public-key Homomorphic Encryption

1. Secret key: an even number \( p \)

   Public key: \( \left[ q_0 p + 2r_0, q_1 p + 2r_1, \ldots, q_t p + 2r_t \right] = (x_0, x_1, \ldots, x_t) \)

2. To Encrypt a bit \( b \): pick random subset \( S \subseteq [1 \ldots t] \)

   \[ c = \sum_{i \in S} x_i + 2r + b \mod x_0 \]

   \[ c = p \left[ \sum_{i \in S} q_i \right] + 2 \left[ r + \sum_{i \in S} r_i \right] + b \mod x_0 \]

   \[ = p \left[ \sum_{i \in S} q_i - kq_0 \right] + 2 \left[ r + \sum_{i \in S} r_i - kr_0 \right] + b \]

   (mult. of \( p \)) + ("small" even noise) + \( b \)
Public-key Homomorphic Encryption

Ciphertext Size Reduction

1 Secret key: an $n^2$-bit odd number $p$

Public key: $[q_0p+2r_0, q_1p+2r_1, \ldots, q_mp+2r_m] \overset{\Delta}{=} (x_0, x_1, \ldots, x_t)$

2 To Encrypt a bit $b$: pick random subset $S \subseteq [1\ldots t]$ 

$$c = \sum_{i \in S} x_i + 2r + b \pmod{x_0}$$

3 To Decrypt a ciphertext $c$: 

- $c \pmod{p} = 2 \cdot r + b \pmod{p} = 2 \cdot r + b$
- read off the least significant bit

4 Eval: **Reduce mod $x_0$ after each operation**
Public-key Homomorphic Encryption

Ciphertext Size Reduction

1. Secret key: an $n^2$-bit odd number $p$

2. Public key: $[q_0p+2r_0, q_1p+2r_1, \ldots, q_tp+2r_t] \overset{\Delta}{=} (x_0, x_1, \ldots, x_t)$

3. To Encrypt a bit $b$: pick random subset $S \subseteq \{1, \ldots, t\}$

- Resulting ciphertext $< x_0$
- Underlying bit is the same (since $x_0$ has even noise)
- Noise does not increase by much(*)

\[ c(b) = \left( \sum_{i \in S} q_ip + 2r_i \right) \pmod{p} = 2r + b \]

- read off the least significant bit

4. Eval: Reduce mod $x_0$ after each operation

(*) additional tricks for mult
A Roadmap

- **Secret-key “Somewhat”** Homomorphic Encryption
- **Public-key “Somewhat”** Homomorphic Encryption
- **Public-key FULLY** Homomorphic Encryption
How “Somewhat” Homomorphic is this?

Can evaluate (multi-variate) polynomials with \( m \) terms, and maximum degree \( d \) if \( d \ll n \).

\[
m \cdot 2^{nd} < \frac{p}{2} = \frac{2^{n^2}}{2} \quad \text{or} \quad d \sim n
\]

\[
f(x_1, \ldots, x_t) = x_1 \cdot x_2 \cdot x_d + \ldots + x_2 \cdot x_5 \cdot x_{d-2}
\]

\( m \) terms

Say, noise in \( \text{Enc}(x_i) \) < \( 2^n \)

Final Noise \( \sim (2^n)^d + \ldots + (2^n)^d = m \cdot (2^n)^d \)
Bootstrapping: from “somewhat HE” to “fully HE”
Bootstrapping: from “somewhat HE” to “fully HE”

**Theorem [Gentry’09]**: Convert “bootstrappable” → FHE.

FHE = Can eval all circuits

Decrypt-then-NAND circuit

```
Dec ---- NAND ---- Dec
  c_1  sk  c_2  sk
```
Is our Scheme “Bootstrappable”?

What functions can the scheme evaluate?

( polynomials of degree < n )

\( \supseteq \)

Complexity of the Decrypt-then-NAND circuit

(degree \( \sim n^{1.73} \) polynomial)

Can be made bootstrappable by “preprocessing” some of the decryption outside the decryption circuit (Following [Gentry 09])

Caveat: Assume Hardness of “Sparse Subset Sum”
Security
(of the “somewhat” homomorphic scheme)
The Approximate GCD Assumption

**Parameters of the Problem:** Three numbers $P, Q,$ and $R$

**Assumption:** No PPT adversary can guess the number $p$

- $p \leftarrow [0 \ldots P]$ (odd $p$)
- $q_{1} \leftarrow [0 \ldots Q]$ (even $q_{1}$)
- $r_{1} \leftarrow [-R \ldots R]$ (even $r_{1}$)
- $(q_{1}p + r_{1}, \ldots, q_{t}p + r_{t})$
Assumption: no PPT adversary can guess the number $p$

(proof of security)

Semantic Security [GM’82]: no PPT adversary can guess the bit $b$

$$PK = (q_0p+2r_0, \{q_ip+2r_i\})$$

$$Enc(b) = (qp+2r+b)$$
Progress in FHE

► “Galactic” → “Efficient”

Asymptotically: nearly \textit{linear-time}\(^*\) algorithms

Practically:

- Implementations, including bootstrapping and “packing”
  github.com/shaih/HElib  github.com/lducas/FHEW
- a few milliseconds for Enc, Dec [LNV’11,Gentry Halevi Smart ‘11]
- a few minutes (amortized) for evaluating an AES block [GHS ‘12]
- single bootstrapping < 1 second [Ducas Micciancio ’14]
- bootstrapping takes 5.5 minutes and allows a “payload” of depth 9 computation on \(GF(2^{16})^{1024}\) vectors

Strange assumptions → Mild assumptions

- Best Known [BGV11]: (leveled) FHE from worst-case hardness of \(n^{O(\log n)}\)-approx short vectors on lattices

*linear-time in the security parameter
Multi-key FHE

\[ c_1 = \text{Enc}(pk_1, x_1) \]

\[ c_2 = \text{Enc}(pk_2, x_2) \]
Multi-key FHE

\[ y = \text{Eval}(f, c_1, c_2) \]

Correctness:
\[ \text{Dec}(sk_1, sk_2, y) = f(x_1, x_2) \]
Fully Homomorphic Encryption

Whiteboard discussion:
• Properties
• Performance
• Contrast with obfuscation
• Usefulness
Protecting memory using Oblivious RAM
Motivation: memory/storage attacks

- **Physical attacks**
  - Memory/storage is on a physical separate device (DRAM chip, SD card, hard disk, …)
  - Communication between CPU and device is easy to tap
  - Memory/storage device may be under attack or stolen
    - Aggravated by data remanence problem

- **Software side channels**
  - Leakage of accesses memory addresses across software confinement boundaries (via data cache, instruction cache, page table, …)

- **Network attacks**
  - External storage (file server, Network Attached Storage, cloud service, …)
  - Remote server/appliance/provider may be compromised
Protecting against memory attack

• Computation model:
  – Random access memory
  – Small processor (logarithmic in memory size)

• Leakage/tampering model:
  – All memory accesses (both data and address) leak or are corrupted arbitrary (relaxation: by polytime adversary)
  – Processor assumed secure

• Goal: a compiler that converts any program into one that resists memory attacks
  – Functionality: input/output precisely preserved
  – Security: privacy against leakage [MR04] with suitable (restricted) circuit classes and admissible functions
Protecting memory content from leakage

- Encrypt the whole memory as a single message
- Encrypt every block separately
  - encrypt block data using AES
  - encrypt block number + data using AES
  - encrypt block using semantically-secure (probabilistic encryption)
- Keep the decryption key inside the secure processor
Protecting memory content from corruption

- Sign every block, keep the signing key inside the secure processor
- Hash every block, keep digests inside the secure processor
- Using Merkle trees
  - Maintain a Merkle hash tree over the memory
  - Merkle nodes stored in the untrusted memory
  - Merkle root stored in secure processor
  - At every read, processor verifies Merkle path
  - At every write, update Merkle path
Compile any program $P$ and memory size $n$ into a new program $P'$, such that: (this definition follows [Chung Pass 2013])

For any $P$ with memory size $n$, and input $x$:

- **Correctness:** $P'(x) = P(x)$ (up to some small failure probability)
- **Efficiency:**
  - $P'$ on $x$ runs $c(n)$ times longer than $P$ on $x$, where $c(\cdot)$ is the computational overhead
  - $P'$ uses memory of size $m(n) \cdot n$, where $m(\cdot)$ is the memory overhead
  - Extra registers in secure processor
- **Obliviousness (security):**
  For any $P_1, P_2$ with memory size $n$, and inputs $x_1, x_2$, such that the number of memory accesses done by $P_1$ on $x_1$ is the same as $P_2$ on $x_2$, the $(\text{address}, \text{val})$ memory transcript of $P'_1$ on $x_1$ is statistically close to that of $P'_2$ on $x_2$. 
“Simple ORAM” construction

[Chung Pass ‘13]

Given a program $P$ and memory size $n$, output $P'$:

$P'$ proceeds like $P$, except:

- $\text{read}(r) \mapsto \text{Oread}(r)$
- $\text{write}(r, val) \mapsto \text{Owrite}(r, val)$
- Memory divided into blocks of size $\alpha$.
- External memory holds a complete binary tree of depth $d = \log\left(\frac{n}{\alpha}\right)$
- $Pos$ maps each memory blocks $b$ to a leaf $pos$.

Invariant: the content of block $b$ is stored somewhere along path to $pos$.

- Each node contains a bucket: at most $K$ tuples $(b, pos, data)$ where $b$ is a block index and $v$ is the block’s data.
  ($K = \text{polylog}(n)$)
- All registers and memory are initialized to $\bot$. 
Simple ORAM” construction: reading

**Oread**(*r*):

- *b* is *r*’s block
- *pos* ← *Pos*[*b*]
- **Fetch** *r*’s block by traversing path from root to *pos* looking for a tuple (*b*, *pos*, *v*). (if not found, output ⊥)
- **Update map** *Pos*[*b*] ← *pos’* chosen at random.
- **Put back** (*b*, *pos’*, *v*) into the root’s bucket. (if overflow, output ⊥)
- **Flush** tuples down a path to a random *pos**, as far as they can go while consistent with invariant. (if overflow, output ⊥)

**Obliviousness**: each *Oread* operation traverses the tree along two paths that are chosen at random and independently of the history so far (doing a single read and single write at every node).
Simple “ORAM” construction: further details

- **Writing:**
  \[ \text{Owrite}(r, val) \]  
  same as \( \text{Oread}(r) \) except we put back the updated \((b, pos', v')\).

- **Storing the position map**
  - Problem: the position map is too large.
  - Solution (“full-fledged construction”):
    recursively stored the position map in a smaller oblivious RAM (same \( K \) but smaller memory).

- **Correctness:**
  Obvious as long as overflows don’t happen. Easy probabilistic analysis shows that overflows happen with negligible probability (for suitable parameters \( \alpha \) and \( K \)). See [Chung Pass ’13 – “A Simple ORAM”] for details.

- **Overheads:** all polylogarithmic. \( O(1) \) registers suffice.

**Other ORAMs**

- Lower bound: \( \log(n) \) computational overhead.
- There are several variants of such “path ORAM”, and others.
- Implemented in software, FPGA hardware.