CPSC 367: Cryptography and Security

Instructor: Michael Fischer
Lecture by Jaspal Singh

Lecture 23
April 18, 2019

Homomorphic Encryption

The Millionaires' Problem

Secure Multiparty Computation
- Definition and Applications
- A Simple Secure Sum Protocol

Generic Secure computation
- Generic MPC Using Value Shares

Homomorphic Encryption

Goals of encryption

The main goal of encryption is to provide data confidentiality.

Normally, there is a lot you can do with your unencrypted data: analyze, search, compute, etc.

However, once data is encrypted there is not much you can do with it.

Encrypted data $\rightarrow$ secured and useless
Unencrypted data $\rightarrow$ unsecured and useful

Working with encrypted data

Solution: Encrypt – decrypt – perform operations – re-encrypt

Problems: Can get very expensive very quickly. Privacy issues.

Another solution: Perform at least *some* operations on encrypted data *without* decrypting it.

Problems: How do we do that? What operations should be allowed? Will it affect security properties of the encryption scheme?

Homomorphic encryption

Informally, homomorphic encryption is an encryption scheme with a special property that allows operations applied to ciphertext be preserved and carried over to the plaintext.

Types of homomorphism

An encryption scheme can be homomorphic with respect to one or more group operators.

An encryption scheme is *additively* homomorphic if we consider the addition operator, and *multiplicatively* homomorphic if we consider the multiplication operator.

Types of homomorphic encryption

Partially homomorphic encryption – it is possible to perform operations on encrypted data with respect to one group operator. For example, from $E(x)$, $E(y)$ compute $E(x+y)$ but not $E(x*y)$.

Fully homomorphic encryption – it is possible to perform operations on encrypted data with respect to two group operator. For example, from $E(x)$, $E(y)$ compute $E(x+y)$ and $E(x*y)$.

Somewhat homomorphic encryption – it is possible to perform a limited number of operations on encrypted data with respect to two group operators. For example, we can only evaluate low-degree polynomials over encrypted data.

Applications of homomorphic encryption

- Cloud computing (untrusted third parties can be used)
- E-voting (votes can be counted without revealing what they are)
- Private information retrieval (searching encrypted databases)

Partially homomorphic encryption schemes

There are many encryption schemes which have the desired homomorphic property.

You should be familiar with at least some of them:

- RSA (multiplicatively)
- ElGamal (multiplicatively)
- Exponential Elgamal (additively for small numbers)
- Goldwasser-Micali (additively)

(Plain) RSA

Public key: (e, N)
Private key: (d, N)
Encryption function: $E(m) = m^e \bmod N$

Multiplicatively homomorphic property:

$$\begin{aligned} E(m_1) * E(m_2) &= (m_1^e \bmod N) * (m_2^e \bmod N) \bmod N \\ &= m_1^e * m_2^e \bmod N \\ &= (m_1 * m_2)^e \bmod N \\ &= E(m_1 * m_2) \end{aligned}$$

ElGamal

Public key: (p, g, b), where $b = g^x$
Private key: (x)
Encryption function: $E(m) = (g^r, m * b^r)$ for a random $r \in Z_{\phi(p)}$

Multiplicatively homomorphic property:

$$\begin{aligned} E(m_1) * E(m_2) &= \\ (g^{r_1}, m_1 * b^{r_1})(g^{r_2}, m_2 * b^{r_2}) &= \\ (g^{r_1+r_2}, (m_1 * m_2) b^{r_1+r_2}) &= \\ E(m_1 * m_2) & \end{aligned}$$

Exponential ElGamal

Public key: (p, g, b), where $b = g^x$
Private key: (x)
Encryption function: $E(m) = (g^r, g^m * b^r)$ for a random $r \in Z_{\phi(p)}$

Additively homomorphic property:

$$\begin{aligned} E(m_1) * E(m_2) = \\ (g^{r_1}, g^{m_1} * b^{r_1})(g^{r_2}, g^{m_2} * b^{r_2}) = \\ (g^{r_1+r_2}, g^{(m_1*m_2)} b^{r_1+r_2}) = \\ E(m_1 + m_2) \end{aligned}$$

Q) Why does this scheme work only for small messages m?

Fully homomorphic encryption

The first fully homomorphic encryption scheme using lattice–based cryptography was presented by Craig Gentry in 2009.[1]

Later in 2009 a second fully homomorphic encryption scheme which does not require ideal lattices was presented.[2]

A lot of changes since then.

[1] C. Gentry, *Fully Homomorphic Encryption Using Ideal Lattices*, STOC 2009

[2] M. van Dijk, C. Gentry, S. Halevi and V. Vaikuntanathan *Fully Homomorphic Encryption over the Integers*, Eurocrypt 2010

FHE performance

Gentry estimated[3] that performing a Google search with encrypted keywords would increase the amount of computing time by about a trillion. Moore's law calculates that it would be 40 years before that homomorphic search would be as efficient as a search today.

At Eurocrypt 2010, Craig Gentry and Shai Halevi presented a working implementation of fully homomorphic encryption.
Martin van Dijk about the efficiency:

> "Computation, ciphertext-expansion are polynomial, but a rather large one..."

[3] *IBM Touts Encryption Innovation. New technology performs calculations on encrypted data without decrypting it* computerworld.com, M. Cooney,

Current FHE efforts

FHE is a very popular research area. Three main directions:

1. Improving the scheme itself (security)
   - Relying on standard hardness assumptions
2. Improving the bootstrapping phase (efficiency)
   - Leveled FHE schemes that are initialized with a bound on the maximal evaluation depth
3. Implementations
   - HElib[4], a software library that implements homomorphic encryption.
   - Extension of HElib that includes the full bootstrapping phase[5]

[4] GitHub Repository

[5] S. Halevi and V. Shoup. *Bootstrapping HElib*, 2014

Security of homomorphic encryption

Let's (informally) rephrase what homomorphic encryption is.

"If you encrypt some plaintext using homomorphic encryption, then by changing the ciphertext you can change the corresponding plaintext".

Q: Is it a good or bad property?

Security of homomorphic encryption

Non–malleability is a desirable security goal for encryption schemes so that the attacker cannot tamper with the ciphertext to affect the plaintext and go undetected.

However, homomorphic encryption implies malleability!

To reconcile this situation, we want an encryption scheme to be non-malleable except for some desired operations.

However, it's difficult to capture the notion of "some malleability allowed."[6]

[6] B. Hemenway and R. Ostrovsky, *On Homomorphic Encryption and Chosen-Ciphertext Security*, PKC 2012

The Millionaires' Problem

The Millionaires' Problem

The Millionaires' problem, introduced by Andy Yao in 1982, began the study of *privacy-preserving multiparty computation*.

Alice and Bob want to know who is the richer without revealing how much they are actually worth.

Alice is worth I million dollars; Bob is worth J million dollars.

They want to determine whether or not $I \geq J$, but at the end of the protocol, neither should have learned any more about the other person's wealth than is implied by the truth value of the predicate $I \geq J$.

Secure Multiparty Computation

Secure multiparty computation (MPC)

Consider n parties $P_1, P_2, \ldots, P_n$ with private inputs $x_1, x_2, \ldots, x_n$ and a public function f.

The MPC protocol must output $f(x_1, x_2, \ldots, x_n)$ while preserving certain *security properties*, even if some of the parties collude and maliciously attack the protocol

Normally, this is modeled by an external adversary $\mathcal{A}$ that may corrupt some parties and coordinates their actions

Applications

- Private Auction
  - Inputs: bids
  - outputs: winning party and winning price
- Anonymous credentials
  - Inputs: credential with personal information, e.g. driving license
  - Output: proof of a certain property of the credentials, e.g. age over 21

Applications

- Disease tests with DNA sequences
  - Inputs: test algorithm, sequenced genome
  - Output: test result
- Data sharing between hospitals
  - Inputs: hospital databases information, e.g. driving license
  - Output: common patients

Some commonly studied security properties

- Correctness: parties obtain correct output (even if some parties misbehave)
- Privacy: only the output is learned (nothing else)
- Independence of inputs: parties cannot choose their inputs as a function of other parties' inputs
- Fairness: if one party learns the output, then all parties learn the output
- ...

The Adversary in an MPC protocol

One could make various assumptions about the power of the adversary. Generally we model the adversary to have control over the data and actions of a set of t parties (also known as corrupt parties).

Most commonly studied adversarial models:

- Semi-honest adversarial model - adversary only observes the *views* of the corrupt parties
- Malicious adversarial model - adversary controls the behavior of the corrupt parties during the protocol

Defining privacy requirement for the semi-honest model

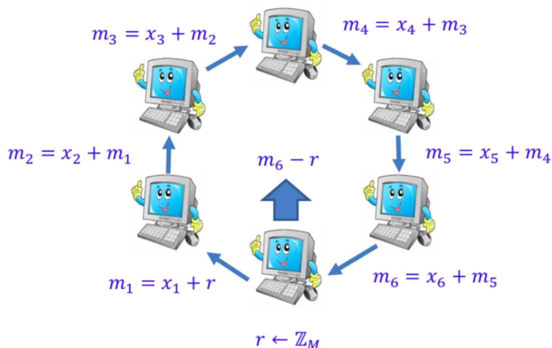
There exist an efficient algorithm S that can simulate the views of the corrupt parties using only the input and output of the corrupt parties.

Let C be the set of corrupt parties, y_i and $view_i$ represent the output and the view of party P_i in the protocol respectively. Then,

$$S(\{x_i, y_i\}_{i \in C}) \equiv \{view_i\}_{i \in C}$$

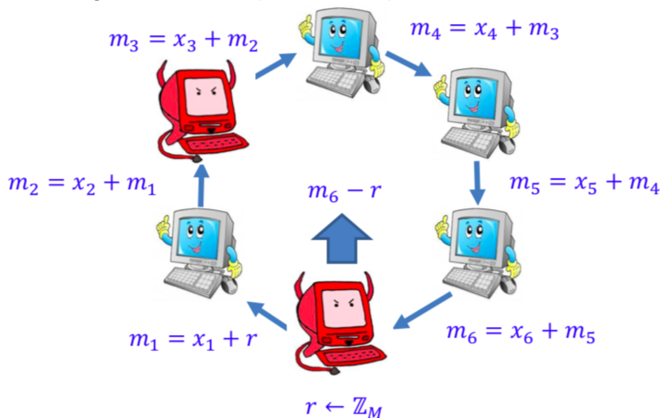
A simple Secure Sum Protocol

Party P_i inputs $x_i \in \mathbb{Z}_m$ for some m.
Output: $(\sum_{i=1}^{n} x_i \mod m)$
Secure against a single corrupt party?



A simple Secure Sum Protocol

Secure against a corruption of 2 parties?

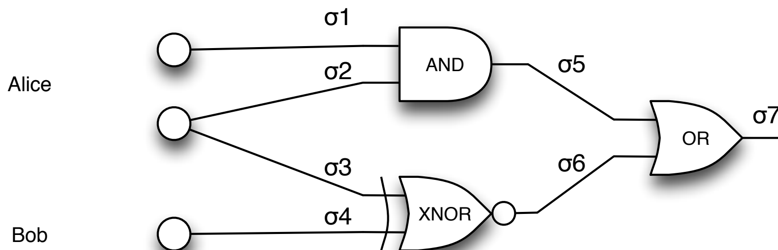


Generic Secure computation

Boolean functions computed by circuits

Let $\bar{z} = f(\bar{x}, \bar{y})$, where $\bar{x}$, $\bar{y}$, and $\bar{z}$ are bit strings of lengths n_x, n_y, and n_z, respectively, and f is a Boolean function computed by a polynomial size Boolean circuit C_f with $n_x + n_y$ input wires and n_z output wires.

Example:



Private evaluation

In a *private evaluation* of C_f, Alice furnishes the (private) input data to the first n_x input wires, and Bob furnishes the (private) input data for the remaining n_y input wires. The n_z output wires should contain the result $\bar{z} = f(\bar{x}, \bar{y})$. The corresponding *functionality* is

$$F(\bar{x}, \bar{y}) = (\bar{z}, \bar{z}).$$

Alice and Bob should learn nothing about each other's inputs or the intermediate values of the circuit, other than what is implied by their own inputs and the output values $\bar{z}$.

Circuit evaluation

An evaluation of a circuit assigns a Boolean value σ_w to each wire of the circuit. The input wires are assigned the corresponding input values.

Let G be a gate with input wires u and v and output wire w that computes the Boolean function $g(x, y)$. In a correct assignment, $\sigma_w = g(\sigma_u, \sigma_v)$.

A complete evaluation of the circuit first assigns values to the input wires and then works its way down the circuit, assigning a value to the output wire of any gate whose inputs have already received values.

Generic MPC Using Value Shares

Value shares

In a private evaluation using *value shares*, we split each value σ_w into two random shares a_w and b_w such that $\sigma_w = a_w \oplus b_w$.

- Alice knows a_w; Bob knows b_w.
- Neither share alone gives any information about σ_w, but together they allow σ_w to be computed.

After all shares have been computed for all wires, Alice and Bob exchange their shares a_w and b_w for each output wire w.

They are both then able to compute the circuit output.

Obtaining the shares

We now describe how Alice and Bob obtain their shares while maintaining the desired privacy.

There are three cases, depending on whether w is

1. An input wire controlled by Alice;
2. An input wire controlled by Bob;
3. The output wire of a gate G.

Alice's input wires

1. Input wire controlled by Alice:

   Alice knows σ_w.

   She generates a random share $a_w \in \{0, 1\}$ for herself and sends Bob his share $b_w = a_w \oplus \sigma_w$.

Bob's input wires

2. Input wire controlled by Bob:

   Bob knows σ_w.

   Alice chooses a random share $a_w \in \{0, 1\}$ for herself.

   She prepares a table T:

σ	$T[\sigma]$
0	a_w
1	$a_w \oplus 1$.

   Bob requests $T[\sigma_w]$ from Alice via OT_1^2 and takes his share to be $b_w = T[\sigma_w] = a_w \oplus \sigma_w$.

Obtaining shares for gate output wires

3. Output wire of a gate G:
   Let G have input wires u, v and compute function $g(x, y)$.
   Alice chooses random share $a_w \in \{0, 1\}$ for herself.
   She computes the table

$$
\begin{array}{rcl}
T[0,0] & = & a_w \oplus g(a_u, a_v) \\
T[0,1] & = & a_w \oplus g(a_u, a_v \oplus 1) \\
T[1,0] & = & a_w \oplus g(a_u \oplus 1, a_v) \\
T[1,1] & = & a_w \oplus g(a_u \oplus 1, a_v \oplus 1)
\end{array}
$$

   (Equivalently, $T[r, s] = a_w \oplus g(a_u \oplus r, a_v \oplus s)$.)

   Bob requests $T[b_u, b_v]$ from Alice via OT_1^4 and takes his share to be $b_w = T[b_u, b_v] = a_w \oplus g(\sigma_u, \sigma_v)$.

Remarks

1. Alice and Bob's shares for w are both independent of σ_w.
   - Alice's share is chosen uniformly at random.
   - Bob's share is always the XOR of Alice's random bit a_w with something independent of a_w.
2. This protocol requires n_y executions of OT_1^2 to distribute the shares for Bob's inputs, and one OT_1^4 for each gate.[7]
3. This protocol assumes semi-honest parties.
4. Bob does not even need to know what function each gate G computes. He only uses his private inputs or shares to request the right line of the table in each of the several OT protocols.

[7] Note: The n_y executions of OT_1^2 can be eliminated by having Bob produce the shares for his input wires just as Alice does for hers. Our approach has the advantage of being more uniform since Alice is in charge of distributing the shares for all wires.