**2. Show that (a) textbook RSA is multiplicatively homomorphic. (b) CAESAR cipher is homomorphic with respect to concatenation operation**

RSA

Homomorphic scheme: permits user to perform computation on data that has been encrypted, without having to decrypt it first. When the output is decrypted, the result is the same as the one that would have been produced by doing computations on decrypted data.

Textbook RSA scheme is usually OWA – CPA secure multiplicative homomorphic. The RSA scheme is homomorphic because we can take two values and encrypt them; the decrypted result will be the integer division of the two values. We use the Euclidean function to find the inverse mod N value of the ciphertext which performs the division. RSA can therefore multiply and divide integers by operating on ciphers.

C = ciphertext, V = value

Encrypt

𝑐=𝑚^𝑒 mod𝑁

Decrypt

m = c^d modN

C1 = V^a (mod N)

C2 = V^b (mod N)

C1 : C2 = V^a : V^b (mod N) = (V^a \* (V^b)^{-1}) (mod N)

(V^b)^{-1}) is the inverse of V^b (mod N)

Caesar Cipher

Caesar Cipher is partially homomorphic because it is not necessary to first decrypt the ciphertext before the concatenation operation; for this Caesar Cipher is homomorphic in respect of the concatenation operation: we can perform some operations on the input ciphertext which will produce new ciphertext. Once decrypted the new ciphertext, it will produce an output plaintext corresponding to a desired operation on the input plaintext.

Pseudocode

C1 = Encrypt(13, “Hello”)

C2 = Encrypt(13, “World”)

C3 = Concatenate(C1, C2)

P = Decrypt(13, C3)

* *Fully homomorphic if it supports enough homomorphic operations to implement any function we need –*

**3. In the lecture you have seen how to garble AND gate using a table with 4 rows. This contributes to the cost of MPC.**

**(a) Can you propose a method to reduce the cost i.e. the reduce the number of rows?**

**(b) How many rows do you need for OR gate? Can you reduce the cost (no. of rows)?**

Row reduction: optimization reduces number of rows from 4 to 3

OR gate, like AND gate, can have any number of individual inputs. Commercially OR gates are available in 2, 3, 4 inputs type. Additional inputs are possible cascading together.

OTs needed only for AND gate, if I use OR gate no OT needed. If it’s a two party GMW with XOR, there is no need for OTs. XOR gates are also free.

**6. Secret sharing is extensively used in MPC protocols:**

**(a) Can you describe some of the most popular Secret sharing techniques?**

**(b) Describe how secret sharing is useful as a stand-alone primitive (what are the possible applications) and what purpose it takes when employed in an MPC protocol?**

Splitting a piece of data into multiple parts is known as secret shares. On their own secret shares reveals nothing about the original data. If two parties perform the same operation on a set of shares and then recombine them, it is as that operation was performed on the original data.

ADDITION

Each party adds up their shares and the exchange the results.

Pi (shares: ai & bi) & PJ (shares: aj & bj)

a + b = (ai + bj) + (bi + aj) = (ai + bi) + (aj + bj)

Pi solves ai + bi and only gets one share of b

Pj solves aj + bj and only gets one share of a

MULTIPLICATION

Parties need to communicate during the computation.

a \* b = (ai + aj) \* (bi + bj)

this can be extended to

= (ai \* bi) + ( ai \* bj) + (aj \* bi) + (aj \* bj)

Pi knows ai \* bi and Pj knows aj \* bj.

We need to address the middle part (( ai \* bj) + (aj \* bi)) differently because it would require each party to know the other share: if Pi wants to compute ai \* bj, it would also need to know bj but they know bi and knowing bj would give them knowledge of the value b (and b needs to be kept secret from Pi). To solve this issue we mask a share, introducing a new unknown number to each party; the new unknown number will disappear when the shares are recombined together. We need a third party to generate this unknown numbers that will mask the data that we don’t want to share with the other party.

bj gets masked from Pi and ai gets masked from Pj.

Masks = s, t

Masked valued = alpha, beta

Pi (P1) computes a\*b:

Zi = sti + (si \* beta) + (alpha \* ti) + (alpha \* beta)

Pj (P2) computes a\*b:

Zj = stj + (sj \* beta) + (alpha \* tj)

P3 third party creates three values for masking and then splits them up into shares. The first two numbers are random and the third number is the product of the first two numbers.

And we can subtract those values from the original data like this (P3 sends si and ti to Pi and sj and tj to Pj):

alpha = (ai – si) + (aj – sj) – Pi creates the left side of alpha (ai – si) and beta (aj – ti)

beta = (aj – ti) + (bj – tj) – Pj creates the right side of alpha (aj – sj) and beta (bj – tj)

Pi and Pj can exchange their alpha and beta shares without revealing any information about a and b because the values of a and be are hidden by the values given by P3.

Secret sharing can be used to protect personal data, such as biometric data or private keys or anything else that shouldn’t be public (i.e. missile position). When employed in an MPC protocol secret sharing makes it safe again an attacker , even with unlimited computing power: secret sharing is information theoretically secure as there is a ‘threshold’ that needs to be met in order to reconstruct the secret. If threshold is not met than the secret cannot be reconstructed (i.e. Vanish Computer – University of Washington).

**7. Most of the ML techniques work with floating point arithmetic. Find out how an MPC using integer or binary arithmetic can be applied to work with floating point ML algorithms.**

Some ML applications are limited today due to limitation to data access (i.e. sensible data, personal data, etc.). We need a solution to improve the training data set for ML applications. In this framework MPC can be useful in order to use bigger data set without exposing personal data and preventing unwanted access to data but at the same time improving the data set.

*the Sharemind secure computation framework is capable of executing tens of millions integer operations or hundreds of thousands floating-point operations per second. It can demonstrate robustness in handling a billion integer inputs*

*and a million floating-point inputs in parallel***.**

**Polynomial evaluation**

Polynomial evaluation requires additions. Floating-point additions are expensive due to private shifts. Fixed-point polynomials can be computed much faster.

**8. What is Pallier encryption and how does it work? Show that the text-book Pallier is additively homomorphic.**

Homomorphic encryption allows you to perform operations on encrypted data without needing to decrypt it beforehand.

Pallier encryption is a partial homomorphic encryption which allows two types of operations: addition of two ciphertext and multiplication of a ciphertext by a plaintext number.

Decpriv(addpub(Encpub(m1),Encpub(m2)))=m1+m2

We encrypt m1 and m2 with a public key pub and a private key priv. We get two ciphertexts:

c1 = Encpub(m1)

c2 = Encpub(m2)

Public Key Encryption:

1. generate a pub – priv key pair
2. encrypt m
3. decrypt m

GCD and LCM:

1. gcd(x, y) – outputs the greatest common divisor of x and y
2. lcm (x, y) – outputs the least common multiple of x and y

Key Generation:

1. choose two large prime numbers (p, q) randomly such as gcd(pq, (p -1) (q -1)) = 1. If != 1 start again.
2. calculate n = pq
3. def function L(x) = x-1/n
4. calculate λ as lcm(p – 1, q -1)
5. choose a random integer g in the set ℤ∗ n2 (integers between 1 and n^2)
6. calculate [modular multiplicative inverse](https://en.wikipedia.org/wiki/Modular_multiplicative_inverse) μ=(L(g^λ mod n^2)) ^−1 mod n. If μ does not exist then we need to start again.
7. We can now use the public key (n, g) for encryption
8. We can use the private key λ for decryption

Encryption

1. encryption works for any m as long as thei are in range 0≤m<n
2. pick a random number r in range 0<r<n
3. compute ciphertext c=g^m ⋅ r^n mod n^2

Decryption

1. if the ciphertext has been correctly created using the above encryption process c should be in range 0<c<n^2.
2. Compute the plaintext m = L(c^λ mod n^2) ⋅ μ mod n
3. it is possible to recalculate μ from λ and the public key.

Additively homomorphic encryption:

Addition of two ciphers:

when two ciphers are multiplied the results decrypt to the sum of their plaintexts:

Decpriv(Encpub(m1)⋅Encpub(m2)mod n^2) =m1+m2 mod n

Multiplication of ciphertext with plaintext:

It i salso possible to multiplicate a ciphertext with a plaintext, when a ciphertext is raised to the power of a plaintext. The result decrypts to the product of the two plaintexts:

Decpriv(Encpub(m1)^m2 mod n^2)=m1 ⋅ m2 mod n