COMPUTATION TECHNIQUES FOR ENCRYPTED DATA

Gajendra Deshpande

KLS Gogte Institute of Technology, India

https://github.com/gcdeshpande

Contents

Homomorphic Encryption

- Imagine taking all of your credit card statements and locking them into a safe, to which you have the only key. Your statements are now protected from prying eyes. This is what encryption does.
- But what if you wanted to analyse your expenditure on groceries in the last 12 months? First you would have to unlock the safe and retrieve the statements. So now the documents are out in the open and they can be read by anyone. This is what decryption does.
- The difference with Homomorphic Encryption is that you can create your report without taking the documents out of the safe.

Properties of Homomorphic Encryption

- **Additive Homomorphic Encryption:**

A Homomorphic encryption is additive, if

$Ek(PT1 \oplus PT2) = Ek(PT1) \oplus Ek(PT2)$

As the encryption function is additively homomorphic, the following identities can be described:

The product of two cipher texts will decrypt to the sum of their corresponding plaintexts,

$D(E(m1) \cdot E(m2) \bmod n) = m1 + m2 \bmod n.$

The product of a cipher text with a plaintext raising g will decrypt to the sum of the corresponding plaintexts,

$D(E(m1) \cdot g^{m2} \bmod n2) = m1 + m2 \bmod n.$

Properties of Homomorphic Encryption

- **Multiplicative Homomorphic Encryption:** Homomorphic encryption is multiplicative, if

$Ek(PT1 \otimes PT2) = Ek(PT1) \otimes Ek(PT2)$

- The homomorphic property of the RSA.

Suppose there are two cipher texts, CT1 and CT2.

$CT1 = m1^e \bmod n$

$CT2 = m2^e \bmod n$

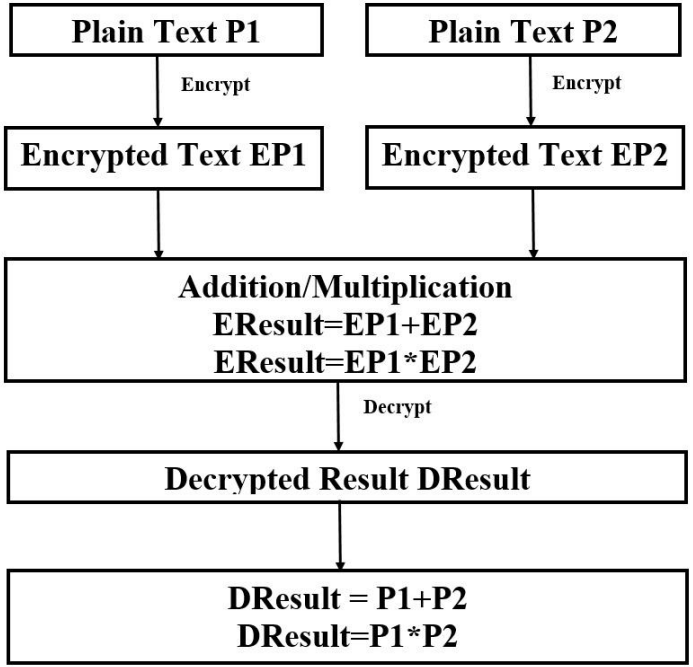
$CT1 \cdot CT2 = m1^e \cdot m2^e \bmod n$

So, multiplicative property: $(m1 \cdot m2)^e \bmod n$

Summary of Homomorphic Properties

Algorithm	Additive	Multiplicative	Applications
RSA	No.	Yes	To secure Internet Banking and credit card transactions
Paillier	Yes	No	E-voting system
ElGamal	No.	Yes	In Hybrid Systems

The Example



- P1=5; P2=10
- EP1=50 EP2=100
- EResult1=50+100=150;
  Eresult2=50*100=5000
- DResult1=15;
  DResult2=50

Experiment with RSA Algorithm

- Selecting two large primes at random: p, q
- Computing their system modulus n=p.q
- Note ø(n)=(p-1)(q-1)
- Selecting at random the encryption key e where $1<e<ø(n)$, $gcd(e,ø(n))=1$
- Solve following equation to find decryption key d
- $e.d=1 \bmod ø(n)$ and $0 \leq d \leq n$
- Publish their public encryption key: pu={e,n}
- Keep secret private decryption key: pr={d,n}

Experiment with RSA Algorithm

- To encrypt a message M the sender:

  obtains public key of recipient pu={e,n}

  computes: $C = m^e \bmod n$, where $0 \leq m < n$

- To decrypt the ciphertext C the owner:

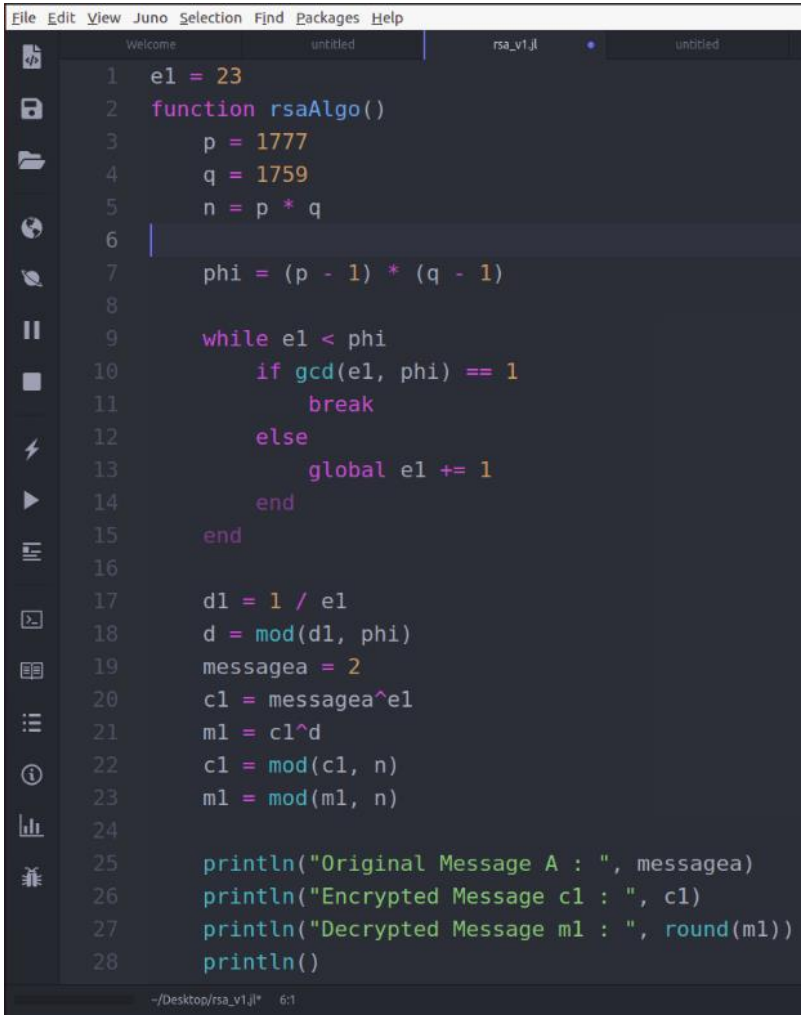
  uses their private key pr={d,n}

  computes: $M = c^d \bmod n$

Experiment with RSA Algorithm

Grade School Method	Karatsuba Method	Fast Fourier Transform
$O(n^2)$	$O(n^{\log_2 3}) = O(n^{1.58})$	$\Theta(n \log(n) \log(\log(n)))$

- The multiplication algorithms were implemented in two steps
- Multiplication algorithms in RSA algorithm were replaced by Karatsuba and FFT methods one after another.
- Multiplication operations were performed on encrypted numbers by Karatsuba and FFT Methods.



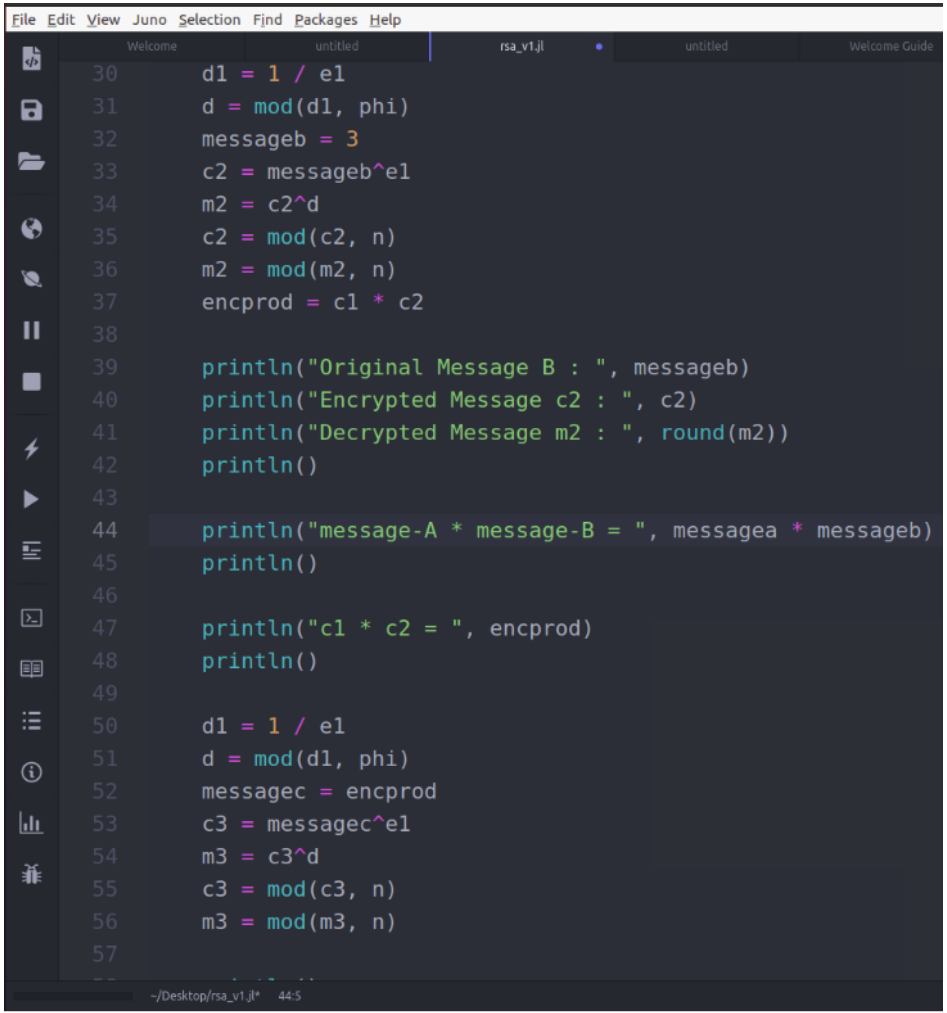
```
e1 = 23
function rsaAlgo()
    p = 1777
    q = 1759
    n = p * q

    phi = (p - 1) * (q - 1)

    while e1 < phi
        if gcd(e1, phi) == 1
            break
        else
            global e1 += 1
        end
    end

    d1 = 1 / e1
    d = mod(d1, phi)
    messagea = 2
    c1 = messagea^e1
    m1 = c1^d
    c1 = mod(c1, n)
    m1 = mod(m1, n)

    println("Original Message A : ", messagea)
    println("Encrypted Message c1 : ", c1)
    println("Decrypted Message m1 : ", round(m1))
    println()
```



```
    d1 = 1 / e1
    d = mod(d1, phi)
    messageb = 3
    c2 = messageb^e1
    m2 = c2^d
    c2 = mod(c2, n)
    m2 = mod(m2, n)
    encprod = c1 * c2

    println("Original Message B : ", messageb)
    println("Encrypted Message c2 : ", c2)
    println("Decrypted Message m2 : ", round(m2))
    println()

    println("message-A * message-B = ", messagea * messageb)
    println()

    println("c1 * c2 = ", encprod)
    println()

    d1 = 1 / e1
    d = mod(d1, phi)
    messagec = encprod
    c3 = messagec^e1
    m3 = c3^d
    c3 = mod(c3, n)
    m3 = mod(m3, n)
```

```
57
58      println()
59      println("Decryption( c1 * c2 ) : ", round(m3))
60      println()
61  end
62
63  @time rsaAlgo()
64
```

~/Desktop/rsa_v1.jl* 64:1

```
Activities  Terminal ▾                                  Jul 15 01:41
gajendra@gajendra-HP-Laptop-15-da0xxx: ~/Desktop
(base) gajendra@gajendra-HP-Laptop-15-da0xxx:~/Desktop$ julia rsa_v1.jl
Original Message A : 2
Encrypted Message c1 : 2137122
Decrypted Message m1 : 2.0

Original Message B : 3
Encrypted Message c2 : 2051153
Decrypted Message m2 : 3.0

message-A * message-B = 6

c1 * c2 = 4383564201666


Decryption( c1 * c2 ) : 6.0

  0.251996 seconds (605.69 k allocations: 28.309 MiB)
(base) gajendra@gajendra-HP-Laptop-15-da0xxx:~/Desktop$
```

File Edit View Juno Selection Find Packages Help

Welcome | untitled | rsa_v1.jl | rsaFFT.jl | untitled | Welcome Guide

```julia
using FFTW
println("x=2027889  y=1774865")

function mulfft()
    x = [9, 8, 8, 7, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0]
    y = [5, 6, 8, 4, 7, 7, 1, 0, 0, 0, 0, 0, 0, 0]
    p = fft(x)
    q = fft(y)
    pq = p .* q
    pqinv = ifft(pq)

    z = []
    for i = 0:length(pqinv)-1
        append!(z, 10^i)
    end

    sum1 = real(pqinv) .* z
    result = sum(sum1)
    println("Multiplication (x*y) using FFT ", result)
    println("Multiplication time using FFT")
end
```

```julia

function mult()
    x = 2027889
    y = 1774865
    result = x * y
    println()
    println("Multiplication (x*y) using * operator ", result)
    println("Multiplication time using * operator")
end

@time mult()
println()
@time mulfft()
```

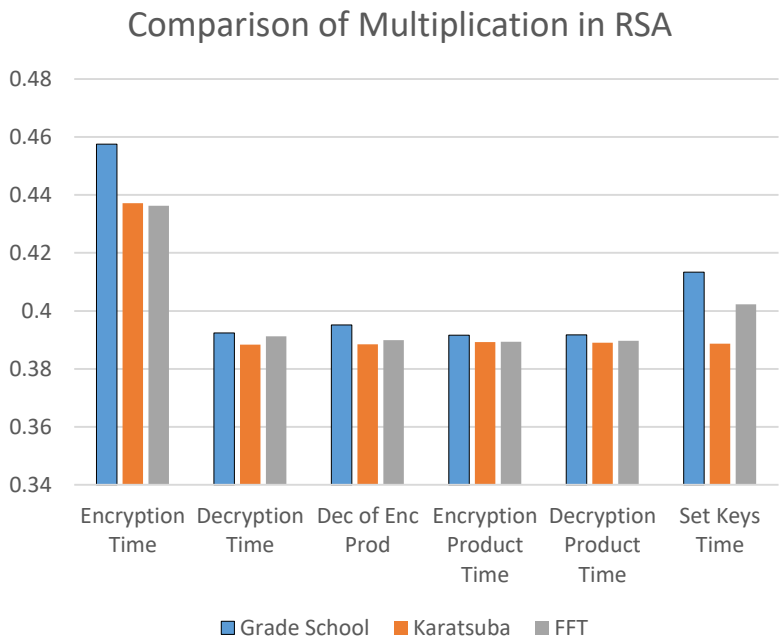
gajendra@gajendra-HP-Laptop-15-da0xxx: ~/Desktop

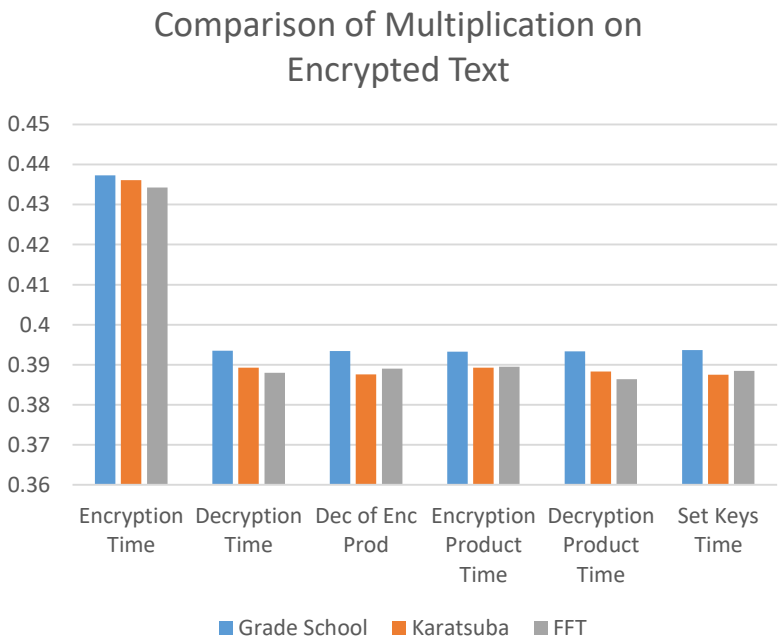
```
gajendra@gajendra-HP-Laptop-15-da0xxx:~/Desktop$ julia rsaFFT.jl
x=2027889       y=1774865

Multiplication (x*y) using * operator 3599229209985
Multiplication time using * operator
  0.017288 seconds (20.09 k allocations: 970.217 KiB)

Multiplication (x*y) using FFT 3.599229209985002e12
Multiplication time using FFT
  0.267845 seconds (698.30 k allocations: 32.888 MiB)
(base) gajendra@gajendra-HP-Laptop-15-da0xxx:~/Desktop$
```

Results


Comparison of Multiplication in RSA
0.48
0.46
0.44
0.42
0.4
0.38
0.36
0.34
Encryption Time
Decryption Time
Dec of Enc Prod
Encryption Product Time
Decryption Product Time
Set Keys Time
Grade School
Karatsuba
FFT


Comparison of Multiplication on Encrypted Text
0.45
0.44
0.43
0.42
0.41
0.4
0.39
0.38
0.37
0.36
Encryption Time
Decryption Time
Dec of Enc Prod
Encryption Product Time
Decryption Product Time
Set Keys Time
Grade School
Karatsuba
FFT

Machine Learning and Homomorphic Encryption

- Pre-processing: Map the numbers in the dataset to random numbers
- Encrypt the data set using cryptographic algorithms such as RSA, paillier or any other cryptosystem
- Perform the computations on encrypted data
- Decrypt the results
- Post-processing: rounding up/down, remap random numbers to original numbers

Conclusion

- Homomorphic Encryption enables computation on untrusted resource. The Computation time over cipher text can be reduced by using Karatsuba or FFT techniques.
- Training and testing machine learning model may involve additional steps such as pre-processing and post-processing and results into additional computational complexity.

Thank You!

Widescreen Test Pattern (16:9)