CS411 HW3 Yağız Gürdamar

Q1-)

The RSA decryption formula is:

M =C^d mod N

However, since we don't have d, we need to rely on the property that M<<N. This suggests that if e is small M^e might still be less than N therefore M^e mod N=M^e. I will try to find M by solving M^e=C.

You can check hw3\_q1.py for code part I used a library for computation.

According to approach leverages the fact that when *M* is significantly smaller than N and e is small (like 17 in this case), the ciphertext C does not wrap around modulo N, allowing us to directly compute the eth root of C to find M. Also exact= true indicates that this is indeed the exact root, that means M^e=C. This means that the decryption was successful and I have correctly retrieved the plaintext M using RCA parameters and the property that M is much smaller than N

M= 340282366920938463463374607431768211456

Q2-)

A-) The security vulnerability you described revolves around the use of greatest common divisor (GCD) in a cryptographic context where prime numbers p and q are used. You mentioned that cp = (kp)e, where cp can be factored by k and p, and that the product of p and q is n, which is known. The essence of the vulnerability lies in the factorization of n into p and q, and cp into multiples of p (repeated e times). Using the extended greatest common divisor algorithm (EGCD) on n and cp reveals p as the common element, and subsequently, q can be found by dividing n by p. This simplifies the decryption process, especially after n has been factored. Essentially, you're pointing out how the encryption can be broken down due to the predictable nature of prime factorization in this setup.

B-)

You can check hw3\_q2.py. What I did is is calculating phi which is (p-1)\*(q-1) and taking inverse modulo of e. Because in the RSA to obtain decryption key we have to obey and apply “e x d = 1 mod(phi(n))” formula.

The result of inverse modulo is d and cipherd = message mod(n). When we cast the bytes to string we obtain: Message: b".eerf t'nsi ohw enoynA ?dlrow siht ni esle gnihtyreve naht erom etah I tahw wonk annaw uoy oD .lliw eerf nwo ym tub gnihton yb denrevog era snoitca yM .od ot dediced I enod ev'I taht gniht elgnis yrevE .eerf ma I" Then I realase that this is reverse of plaintext and I changed it:

I am free. Every single thing that I've done I decided to do. My actions are governed by nothing but my own free will. Do you wanna know what I hate more than everything else in this world? Anyone who isn't free.

Q3-)

Nonlinearity Degree: - The degree of the highest order term determines the nonlinearity degree of a Boolean function. The function is more resilient to linear cryptanalysis the higher the degree of nonlinearity.

- The highest order term in the function provided is 4. Consequently, this function has a nonlinearity degree of 4.

Balance: When a Boolean function generates the same amount of 1s and 0s across its input space, it is said to be balanced. For cryptographic operations to prevent biases that might be exploited, balance is essential.

- We would have to create a truth table or take a more analytical approach for this function. But as we can see, the function simplifies to F = x4, which is unbalanced since it depends on the value of x4 alone, when x1 = x2 = x3 = 0. The function is therefore unbalanced.

Correlation Immunity:

- A function is immune to correlation assaults and is said to have correlation immunity of order m if it does not change when up to m of its input bits are fixed. We must examine this function's behavior when certain inputs are maintained constant in order to ascertain its correlation immunity. But as we can see, the function depends on all four variables, thus adjusting just one of them may have an impact on the result. As a result, this function's correlation immunity is probably low.

Q4-)

You can check hw3\_q4.py for my solution.

Q6-)

You can check hw3\_q6.py for my solution.