DOPE encryption and RSA encryption

Goal

Make a simpler version of this as a proof of understanding of RSA encryption. Call it Doxr Open Protocol Encryption and create a library for it (DOPE library). After some working of the DOPE JavaScript library, the limit is theoretically “unlimited,” versus the prior 1024 bit limit imposed on Number type integers (and by extension, pseudo-randomized numbers which leads to imposing that limit on the entire DOPE encryption system)

Core Variables

These are part of the public and private keys. Only p and q aren’t part of them explicitly, but they do come up, and generating the private keypair should also ensure to include the p and q values, so the CRT works.

[PRIVATE] `p`: True Prime Number
[PRIVATE] `q`: True Prime Number
[PUBLIC] `n`: Found through `n = p * q`
[PUBLIC] `e`: Encryption exponent
[PRIVATE] `d`: Decryption exponent

`e` and `d` MUST satisfy `ed = 1 mod (p - 1) (q - 1)`

`e` and d also should satisfy `ed = 1 mod φ(N)`, where φ represents Euler’s totient function.

Ensure e is coprime with φ(N); this means `gcd(e, φ(N)) = 1`

Ensure that **p** and **q** are at the correct bit size!

Use a primality test (like Miller-Rabin) to verify p and q. Miller-Rabin is PROBABILISTIC, so try the test multiple times to figure out if **q** or **p** is prime or composite.

The public key is a pair, (N, e), the private key pair is (N, d). However, all variables should be part of a private DOPE key, so a public key can be reproduced if lost.

Notes

`φ(N) = (p-1)(q-1)` where phi represents Euler’s totient function.

Usually, a good e value is generated through a hardcoded list, and a d value is computed relative to e.

Private and public key files, for aesthetic reasons, should be stored in a JSON format with the file extension ".dope" (ex. `private.dope` and `public.dope`). It doesn't really matter though this is just for the library

Miller-Rabin Test

As I said, this is a probabilistic test, so you need to do it a couple times with different witness values to make sure what you're getting is right.

Start with the number you're testing, n, and find r, which is n - 1.

Then, divide r by 2 until r isn't even: note, r must be divided by at least once. From there, if the value isn't even, keep going until it is. This new r value is your d value. The amount of times you had to divide is s. Mathematically, this is what you are trying to satisfy:

n - 1 = 2^s * d

Where d is odd.

Next, choose a random a value, which is called the witness. The constraints to a are as such:

2 <= a <= n - 2

Then, compute the value of a new variable, x, to possibly determine the results of the test.

x = a ^ d mod n

x *might* be prime if one of the two cases are true:

If x = 1 mod n, then n passes this test.
If x = n - 1 mod n, then n passes this test.y

If you didn't get a true case, then square x repeatedly:

x = x^2 mod n

If, at any step, x = n - 1 mod n, then n passes this test. You can only repeat *s* times.

If n fails ANY of the a bases, then it is definitely NOT prime. If n passes many tests with different witnesses, then it's probably prime. I say probably because this test is not 100%

accurate; sometimes, there are witnesses that lie about a number being a prime, so you need to redo the test with multiple `a` witnesses within the range.

Chinese Remainder Theorem

To decrypt, instead of running `M = C^d mod n` we can speed up this process by possibly using these decryption steps:

To decrypt, we can compute K and L separately and use the Extended Euclidean Algorithm for the modular inverse of q, which is necessary and speeds up the decryption process by about 4x: for things like strings, this is massive. K and L are then processed to restore M.

To compute K:

K = C^d mod p

L = C ^d mod q

Then combine K and L using the Chinese Remainder Theorem; assume J is the inverse of q, mod p (`q^-1 mod p`).

M = L + q * ((K-L) * J mod p)

Note: J must be positive

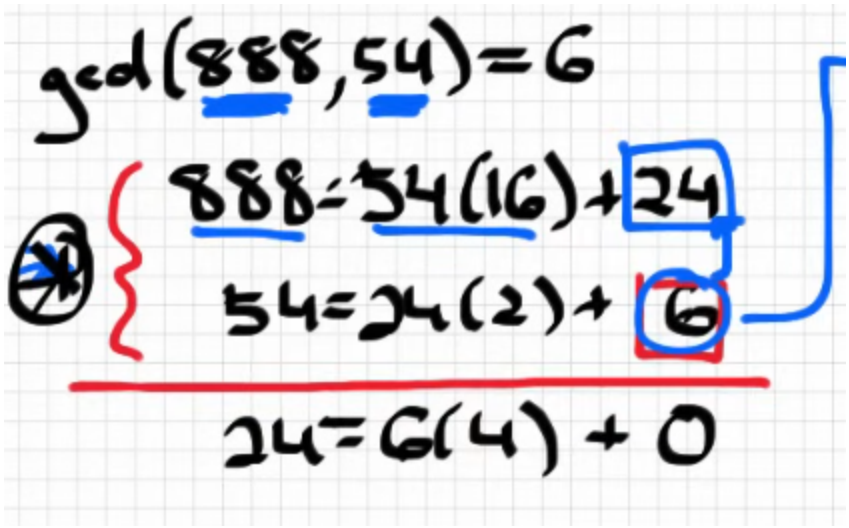
Extended Euclidean Algorithm. It is necessary to use it to find J. J must be positive, if it isn't, add p to it. The two inputs for the original EA function are going to be p and q. More information regarding EEA specifically, upcoming.

Understanding EEA and Complete the Chinese Remainder Theorem

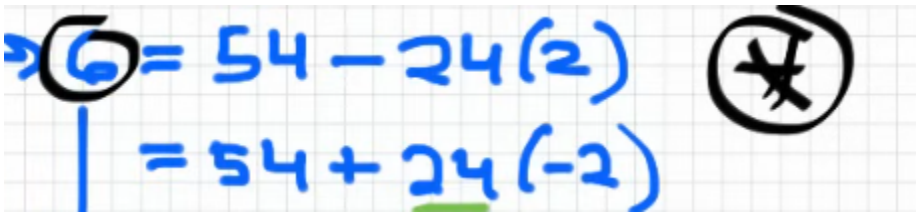
Firstly, to understand EEA, I decided to take a detour and understand the Euclidean Algorithm, as I simply could not grasp the concept of EEA at the start.

It works as such: find the remainder of the bigger of the two provided numbers to find the GCD. When you get the remainder, divide the quotient by the remainder. Repeat until the remainder is 0. Find the last applicable remainder; that is the GCD.

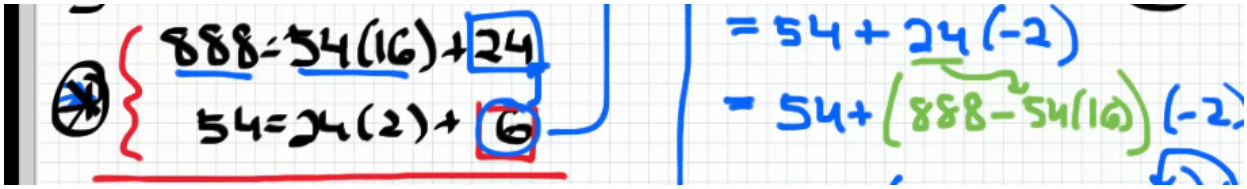
Now, take that solution and the entire equation:



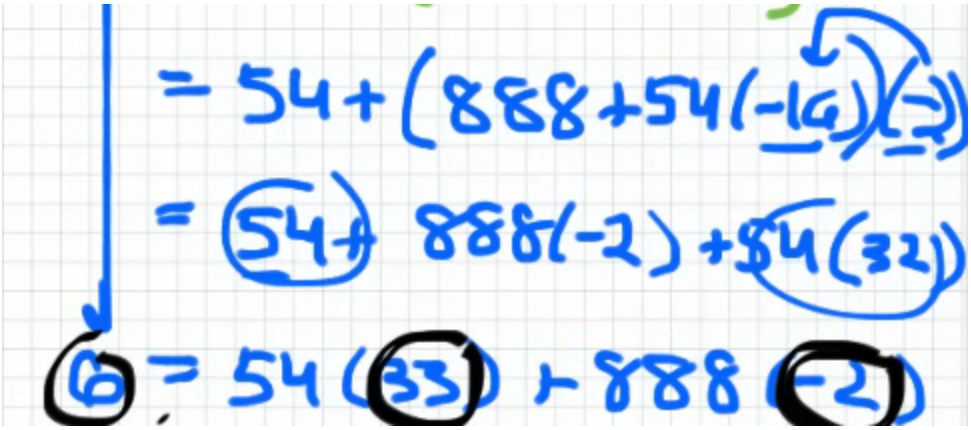
Look at the above image. This is an EA solution, where the GCD between 888 and 54 was 6. Create a new function, by moving the solution equation's terms so that 6 is equal to the rest of the function; then, turn the negative into a positive by moving the -24 (2) to 24 (-2):



See how there's a 24 above the 6 (it was the previous remainder)?



Substitute the 24 in the EEA using the previous equation, 888 = 54 (16) + 24 OR 888 - 54 (16) = 24, and you'll get 54 + (888 - 54(16))(-2) and that can turn into + 54 (-16). From there, multiply everything by -2 and combine like terms, which gets you this:



You can see that you reversed the function. HUGE TIP: Try to write the EEA equation as a linear equation, to make calculations easier.

So what does this have to do with anything? In our case, the 33 here would be the hypothetical J value. Secondly, the GCD, when both primes are inputted, will ALWAYS result in 1, which may be useful for crafting the J value. Essentially, we can do this through tracking coefficients in the program WHILE the EA process is still happening. From there, you can craft the message using the CRT.

Generating d

There are two ways I can do this. This is possible through Fermat's Little Theorem or the EEA, where the EEA is much more efficient. However, Fermat's Little Theorem is simpler.

Fermat's Little Theorem

We can use Fermat's Little Theorem to generate d. It states that that if a (d) is coprime to m (φ(N)), then

$a^{m-1} \equiv 1 \, mod \, m$

So essentially we can simplify this to calculate d, by using the public e encryption exponent:

$d = e^{\varphi(N)-1} \, mod \, \varphi(N)$

The issue with LFT is that it's simply impractical to use such MASSIVE values, so it would be necessary to do something like this:

We keep squaring e until it reaches the value of $e^{\varphi(N)-1}$, but this may take too much time. It's a little theorem that needs many resources.

Extended Euclidean Algorithm

We can compute the value of d using the EEA. See above notes on how to perform it; start with the EA (the original equation should be $gcd(e, \varphi(N))$ and phi N is divided by e), and keep in mind that this is what the end result should look like:

$$e * d + \varphi(N) * k = 1$$

So if the end result of the linear combination looks like this, and it was already determined that e was going to be 17 during the EA:

$$-367 * 17 + 2 * 3120 = 1$$

d would be -367, the coefficient of 17. However, since d cannot be negative, get a positive d by doing this:

$$d = d + \varphi(N)$$

This should make d positive, and you can use this during decryption.

Note: Repeat Encryption

If the library has a function to encrypt and decrypt strings, it should most likely try to encrypt/decrypt the ASCII/number equivalent as little as possible and re-use what has already been encrypted. This should result in less computation power and time necessary to encrypt/decrypt

BASICS of Encryption/Decryption

Above, I go through MUCH MUCH more which makes this process more complicated but also much more efficient and quick.

Mathematically, a value M can be used to generate an encrypted value, C. Only public values are used here!

C = M^e mod N

After all of the decryption

And to decrypt, WITHOUT the CRT or anything else (Please see above!),

M = C^d mod N

Speeding up 2048 bit key generation

Generate a small list of prime numbers (up to 1000) and use a proxy to the Miller-Rabin test. If any of them can divide successfully into the larger number with no remainder, cut the number out and try a different random prime number. This will eliminate a lot more numbers before Modular Exponentiation.

Important: Limitations

This is not a limitation with DOPE, but rather a limitation with RSA as a whole because of how modular arithmetic works:

M < N

This is because once M overtakes N, the remainder resets, leaving you with the wrong thing. So how do we get past this issue?

1. Chunking:

This is the simplest. Chunk the message in case M overtakes N, and ensure the chunked content can be un-chunked by the library.

2. Padding:

I am not familiar with this, nor am I going to try to implement this using the DOPE library.

TODO:

- Implement Montgomery Multiplication for Modular Exponentiation/modPow

Modular Exponentiation

Although as of time of writing I can't explain Modular Exponentiation logically, it aims to solve these types of calculations:

$$x = b^c \bmod t$$

Where b and c (or even just one of the two) are absolutely massive. Doing this as-is is a huge issue since computers should not need to calculate those big numbers. Instead, it can be done as such:

1. Declare r as 1 (it is called the result)
2. Calculate a new base, using this:
   $b = b \bmod d$

3. Then, repeat these steps until c is 0:
    - If the exponent is odd, multiply the current base into the result, r, and then assign r to the remainder of t
      $r = (r * b) \bmod t$
    - Then, no matter the exponent, square the base and assign b to the mod of t
      $b = b^2 \bmod d$
    - Then, finally, halve the exponent (c) and check if it's 0.
4. The end result, x, is r, the result variable.

Keygen Steps

1. Generate two random odd numbers in the desired bit range.
2. Use the Miller-Rabin test on both numbers to make sure they are true primes.
3. Compute N
4. Compute Euler's Totient Function, φ(N), through `(p - 1) * (q - 1)`; this value is used for d
5. Choose a fairly nice public e value from a list of suitable e values.
6. Make sure d satisfies the e * d equation, this is done through the EEA
7. Output the key pair for the public and private.
    - Public Key Pair: (N, e)
    - Private Key Pair: (N, d) - also include the p and q values for CRT!
8. To decrypt, use the Chinese Remainder Theorem
9. Calculate the inverse q value through the EEA
10. Complete the Chinese Remainder Theorem

End Credits

I made this document with very very little technical knowledge about RSA, and I ended up knowing enough to implement it in Javascript with no other dependencies, and throughout the project, I was implementing more math stuff that I didn't know existed, nor did I plan to implement, to make DOPE more efficient and usable. Because of this, you WILL notice inconsistencies within this document; however, I did try to make information consistent throughout the document. DOPE is very dope in my opinion because I've never tried to make my own public-private key encryption method, only symmetric encryption methods. I don't question that it's vulnerable to some nerdy thing that makes this easier to crack, but DOPE is fundamentally just RSA which is used everywhere so it can't be that bad.

Created by doxr.