PROJECT REPORT: RSA ENCRYPTION

A. Outline

- The goal of my project is to learn about RSA encryption – one of the most widely use encryption technique.
- The project contains the following:
  1. Understanding the basics of RSA
  2. Implementing different version of RSA from scratch.
  3. Breaking RSA using factorisation and chosen ciphertext attack.
  4. Applying RSA into a practical application (end to end encrypted chat).
- This report will contains what I did throughout my project and reflection. When reading this documentation, I hope you can understand the basics of RSA encryption.
- All code could be found at: https://github.com/minhlong08/RSA-encryption
- Video demo playlist: playlist

B. Basics of RSA

1. Introduction

- RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission.
- Created in 1977 by: Ron Rivest, Adi Shamir, Leonard Adleman.
- The most common Asymmetric Encryption algorithm.
- It relies on the mathematical difficulty of factoring large prime numbers.

2. Operation

- The RSA algorithm involves four steps: key generation, key distribution, encryption, and decryption.

a) Key generation

- Choose two large prime number ‘p’ and ‘q’.
- Compute ‘$n = p * q$’.
- Compute Euler’s totient ‘$\varphi(n) = (p-1) * (q-1)$’.
- Choose an encryption exponent ‘e’ that matches these requirements:
  - $1 < e < \varphi(n)$
  - e and $\varphi(n)$ are co-prime which mean that the greatest common divisor of e and $\varphi(n)$ is 1 ($\gcd(e, \varphi(n)) = 1$). This ensure that we can compute ‘d’.
- Compute the decryption exponent ‘d’ such that $d \equiv e^{-1} \bmod \varphi(n)$. This means that if we take the product of ‘e’ and ‘d’ and divide by ‘$\varphi(n)$’, we get 1 as the result.
- Our public key pair: (e, n).
- Our private key pair: (d, n).

b) Key distribution

- The sender obtains the recipient's public key *(e, n)*, which can be safely shared with anyone.
- The sender encrypts the message using the public key so only the matching private key can decrypt it.
- The recipient uses their private key *(d, n)* to decrypt the message, which no one else can do without knowing *d*.

c) Encryption

- The sender converts the message into a number M smaller than n.
- They compute the ciphertext C = M^e mod n using the recipient's public key (e, n).

d) Decryption

- The recipient receives the ciphertext C.
- They compute the original message M = C^d mod n using their private key (d, n).

C. Implementing RSA

- To better understand RSA, I implement the RSA algorithm using Python.
- For our RSA encryption, the key functions are: key generation, encryption, and decryption.

1. Simple version

a) Key generation

- Checking if a number is prime

```
def is_prime(self, number):
        if number < 2:
            return False
        if number in (2, 3):
            return True
        if number % 2 == 0:
            return False
        for i in range(3, int(math.sqrt(number)) + 1, 2):
            if number % i == 0:
                return False
        return True
```

- In this version we will check for primality using the definition of prime number.
- Check for divisibility up to $\sqrt{number}$ to determine primality.
- Noted that 1 is not a prime number.

- Generating a random prime

```
def generate_prime(self, min_value, max_value):
        prime = random.randint(min_value, max_value)
        while not self.is_prime(prime):
            prime = random.randint(min_value, max_value)
        return prime
```

- We will use the Python library random to generate a number between a certain range.
- We check if this randomly generated number is prime. If not, we will generate another random number and continue the process until we get a prime number.

- Calculating the modular inverse (use for calculating the private key *d*)

```
def mod_inverse(self, e, phi):
        for d in range(3, phi):
            if (d * e) % phi == 1:
                return d
        raise ValueError("Mod_inverse does not exist!")
```

- Given two number ‘e’ and ‘φ(n)’, we will need to calculate the modular inverse of ‘e’ with modulo ‘φ(n)’.
- This function uses a brute force approach to find ‘d’ by trying all number from 3 up to ‘φ(n)’.
- A better algorithm would be the **Extended Euclidean Algorithm**, but this works for small values.

- Generating the RSA key pair

```
def generate_keys(self):
        while True:
            p = self.generate_prime(1000, 50000)
            q = self.generate_prime(1000, 50000)
            if p == q:
                continue
            self.n = p * q
            if self.n < 65536:
                continue
            phi_n = (p - 1) * (q - 1)
            self.e = random.randint(3, phi_n - 1)
            while math.gcd(self.e, phi_n) != 1 or not self.is_prime(self.e):
                self.e = random.randint(3, phi_n - 1)
            self.d = self.mod_inverse(self.e, phi_n)
            break
```

- We first generate 2 prime number using of generate_prime method above. In this version, we set the range for our prime number to be in between 1000 to 50000.
- Our RSA algorithm will easily be broken if 2 prime numbers are the same so if our ‘p’ and ‘q’ are the same, we will generate them again
- Compute n = p * q. RSA requires the message M to be smaller than n, or else the encryption M^e mod n will "wrap around," causing different messages to produce the same ciphertext (information loss). In our code, we are currently ensuring that n > 65536 (16 bits) which is enough to encrypt at least all ASCII character.
- Calculate φ(n) = (p - 1) * (q - 1).
- Compute ‘e’ based on the requirement mentioned above. Noted that ‘e’ does not have to be prime but enforcing ‘e’ to be primed is good since this ensure that ‘e’ will be co-prime with ‘φ(n)’.
- Compute ‘d’ using the mod_inverse method above.

b) Encryption

```
def encrypt(self, message, public_key=None):
        if public_key is not None:
            e, n = public_key
        else:
            e, n = self.e, self.n

        message_bytes = message.encode('utf-8')
        ciphertext = [pow(byte, e, n) for byte in message_bytes]
        return ciphertext
```

- In our encryption, we either let the user provide us with a key or use the key that we generate above.
- We convert the message string into a sequence of byte values (integers from 0–255) using UTF-8 encoding. Our encryption can safely handle this as we ensure n > 65536.
- For each byte b, it computes the RSA encryption: C = b^e mod n. This transforms each byte into an encrypted number using modular exponentiation.
- Our encrypted message will be an array of number where each number represent a single character

```
Enter your message to encrypt: hello world
Original Message: hello world
Encrypted: [549570775, 324633731, 903447836, 903447836, 542046722, 404313352, 658287686, 542046722, 476062068, 903447836, 368879052]
```

c) Decryption

```
    def decrypt(self, ciphertext, private_key=None):
        if private_key is not None:
            d, n = private_key
        else:
            d, n = self.d, self.n

        decrypted_bytes = bytes([pow(byte, d, n) for byte in ciphertext])
        return decrypted_bytes.decode('utf-8')
```

- For each encrypted integer c, it applies the RSA decryption formula:
  $m = c^d \bmod n$
- This recovers the original byte value (0–255) from the ciphertext.
- It turns the list of decrypted byte values back into a bytes object using the `bytes()` method.
- Then the function converts bytes back to string using UTF-8.
- The function works because the encryption was done byte-by-byte as well. It successfully recovers each character at a time.

The full code can be found at /src/rsa_simple.py

d) Current problems

- I made this version only to understand and demonstrate the logic behind RSA.
- Some noticeable problem with this version:
  - The algorithm we are currently using is largely ineffective and inconsistent, especially in the key generation algorithm (we brute force check a lot). To test this out, I timed the process of key generation.

```
rsa = RSA_SIMPLE()

startime = time.time()
rsa.generate_keys()
endtime = time.time()

runtime = endtime - startime
print(f"Key generation took {runtime:.6f} seconds")

message = input("Enter your message to encrypt: ")
print("Original Message:", message)
```

  - This was the result:

```
src $ python rsa_simple.py
Key generation took 9.350172 seconds
Enter your message to encrypt: hello world
Original Message: hello world
Encrypted: [374895512, 149775928, 133732785, 133732785, 603660322, 70306907, 262502205, 603660322, 169205338, 133732785, 115021
275]
Decrypted: hello world
src $ python rsa_simple.py
Key generation took 0.016062 seconds
Enter your message to encrypt: hello world
Original Message: hello world
Encrypted: [21600796, 31149756, 23216467, 23216467, 15286547, 59806695, 1524715, 15286547, 66036708, 23216467, 32967802]
Decrypted: hello world
```

```
src $ python rsa_simple.py
Key generation took 3.311811 seconds
Enter your message to encrypt: hello world
Original Message: hello world
Encrypted: [721348916, 251534253, 499640801, 499640801, 436884548, 44568172, 259026498, 436884548, 504180787, 499640801, 443478
837]
Decrypted: hello world
src $ python rsa_simple.py
Key generation took 25.881678 seconds
Enter your message to encrypt: hello world
Original Message: hello world
Encrypted: [21462305, 537021941, 696300790, 696300790, 363302205, 242281780, 368162323, 363302205, 249187259, 696300790, 579128
194]
Decrypted: hello world
```

  - As you can see the key generation varies from 0.01 up to 25.8 second (which is 2000 times increase).
  - Also, our key is very small (n is only around 16 bits) which can easily be cracked (this will be demonstrate later). Theoretically our algorithm can generate larger key but due to the current inefficient algorithm, this would take very long.
  - Our public key ‘e’ is sometime very large too which also slow down the encryption speed. Other time our ‘e’ is very small and our encryption can be easier reverse by calculating the roots.

- One of the weak points of RSA encryption is that it is deterministic (the same input always encrypts to the same output). Our current algorithm encrypts byte by byte which even make this worst. With a special input, an attacker can figure out which character map to which numbers.
- For example, with this input, an attacker can figure out the with the current key, letter 'a' maps to '1295113857',letter 'b' maps to '1272925011', and so on. They can do this for the whole alphabet and can break our encryption easily (our encryption essentially right now is a special version of Caesar cipher that turn letters into numbers).

```
src $ python rsa_simple.py
Key generation took 48.249231 seconds
Enter your message to encrypt: abcdefgh
Original Message: abcdefgh
Encrypted: [1295113857, 1272925011, 265812085, 832847840, 1004639556, 621599892, 508008433, 391996969]
Decrypted: abcdefgh
```

2. More efficient version

a) Improving prime generation

- Key generation are the area that we can improve on from the previous version.
- Currently we are checking the if a number is prime naively by checking divisibility from 2 up to sqrt(n) which has time complexity of $O(\sqrt{n})$. This can scale up very quickly with larger number (RSA key need to be very large, up to 2048 bits which is more than 600 digits).
- A better algorithm that is used for prime checking is the Miller-Rabin test which can reduce the time complexity down to $O(klog^3 n)$. This test is better when checking for larger prime number and is commonly used in real world application.
- We can also pre-compute some smaller primes and filter out this smaller prime faster.
- In the updated version, I also want to control the length of the key being generated (bits size). This can be control when we are generating prime numbers 'p' and 'q'

```
    def generate_prime(bits: int) -> int:
        """Generate a random prime number with specified bit length."""
        while True:
            # Generate random odd number with specified bit length
            candidate = random.getrandbits(bits)
            candidate |= (1 << bits - 1) | 1  # Set MSB and LSB to 1

            if MathUtils.is_prime(candidate):
                return candidate
```

- random.getrandbits(bits) generates a random integer between 0 and $2^{bits} - 1$
- However, this number might not use all the bits (the most significant bits might be 0). So, we force the number to have exactly the number of bits we want by setting the most significant bits to 1. Moreover, since prime number cannot be an even number, we also set the least significant bits to 1 to make sure our randomly generated number is odd (which might be prime and save us time of unnecessary checking for an even number to be prime).

Miller-Rabin test

- Miller-Rabin primality test is a probabilistic test for prime number. Noted that this test cannot prove a number is prime but only prove a number is certainly

not prime. So with enough round of testing, we can be highly sure that a number is prime if it passed all the tests.

- Let n be the number we are testing to be prime or not.
- Step 1: choose a random number a.
- Step 2: Checks a certain property of a and n (this condition is guaranteed if n is prime which means if this condition fails n cannot be a prime number).
- If the above property is satisfied, return "could be prime". If not satisfied, return "not prime".
- The crucial idea is that if n is not prime, by repeating this test many times, it is very likely to find a number a that is a witness (prove that n is not prime).

The property of prime number:

- Choose some prime n, write $n = d2^r + 1$ (x is odd).
- Choos random a in the range from 0 to $n - 1$
- If n is prime then we have:

  $$a^d \equiv 1\ (mod\ n) \text{ or } a^{xd} \equiv -1\ (mod\ n) \text{ for some } i\ \in \{0, \dots, r-1\}$$
- If we can find any number a that does not satisfy the above property, we can conclude that n is not prime.
- These properties can be proof using Fermat little theorem but I won't go into it, you can find the full proof in the appendix section.
- How sure can we be about a number is probably prime using this test? If the number n is a composite number, over 75% of the possibly value of a will fails this Miller-Rabin test. So, if we run this test 100 times, the probability that we falsely conclude a composite number as prime will be $6.22 \times 10^{-59}\%$ (incredibly low chance).

b) Improving calculating modular inverse

- We can also improve finding the modular inverse algorithm by using Extended Euclidean Algorithm. Recall that Extended Euclidean Algorithm is an algorithm to compute integer x and y such that

  $$ax + by = \gcd(a, b)$$
- In our case we want to calculate the modular inverse of 'e' mod 'φ(n)'. With Extended Euclidean Algorithm, we can find integers x, y such that

  $$ex + \varphi(n)y = \gcd(e, \varphi(n)) = 1\ (since\ e\ and\ \varphi(n)\ \text{are co} - \text{prime})$$

  Taking modular φ(n) on both side of the equation we will have

  $$ex + 0 \equiv 1\ (\text{mod}\ \varphi(n))$$

  Which means that x is the modular inverse of 'e' mod 'φ(n)'. This is the value we need for 'd'. Therefore, if we run the Extended Euclidean Algorithm on 'e' and 'φ(n)', we can find the modular inverse 'd' that we need.

```
def mod_inverse(a: int, m: int) -> int:
    """Calculate modular inverse of a modulo m."""
    gcd, x, _ = MathUtils.extended_gcd(a, m)

    if gcd != 1:
        raise ValueError("Modular inverse does not exist")

    return (x % m + m) % m
```

The last return line ensure that our modular inverse is not negative.

- Extended Euclidean algorithm can be done by reversing the step of Euclidean algorithm (Euclidean algorithm calculate gcd(a, b) by recursively doing gcd(a,b) = gcd(b, a mod b)).
- Here an example of Euclidean Algorithm to calculate the GCD of 102 and 38

$$102 = 2 \times 38 + 26$$
$$38 = 1 \times 26 + 12$$
$$26 = 2 \times 12 + 2$$
$$12 = 6 \times 2 + 0$$

Therefore the gcd(102,38) = 2

- A code implementation of Euclidean Algorithm would look something like this

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

- Here an example applying Extended Euclidean Algorithm to find x and y such that $102x + 38y = \gcd(102,38) = 2$

We can work backward from the Euclidean algorithm that we calculated

$$2 = 26 - 2 \times 12$$
$$= 26 - 2 \times (38 - 26) = 3 \times 26 - 2 \times 38$$
$$= 3 \times (102 - 2 \times 38) - 2 \times 38 = 3 \times 102 - 8 \times 28$$

Therefore, x = 3 and y = 8

- Here the code implementing Extended Euclidean Algorithm

```
    def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
        """
        Extended Euclidean Algorithm.
        Returns (gcd, x, y) such that a*x + b*y = gcd(a, b).
        """
        if a == 0:
            return b, 0, 1

        gcd, x1, y1 = MathUtils.extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1

        return gcd, x, y
```

- The base case occurs when a == 0, returning (b, 0, 1) because:

$$0 \times 0 + b \times 1 = b$$

- After the recursive call returns gcd, x1, y1 such that:

$$(b \bmod a) \times x1 + a \times y1 = gcd$$

- Rewrite b mod a as:

$$b \bmod a = b - \left\lfloor \frac{b}{a} \right\rfloor \times a$$

- Substitute back and rearrange:

$$a \times \left(y1 - \left\lfloor \frac{b}{a} \right\rfloor x1\right) + bx1 = gcd$$

- Therefore, the new coefficients are:

$$x = y1 - \left\lfloor \frac{b}{a} \right\rfloor x1, y = x1$$

- Our previous version of calculating modular inverse run in O(n) as it brute force all the value up until 'φ(n)'. This updated version has a divide and conquer approach and could be run in O(logn) which is much faster for larger input.

c) Improving choosing public exponent

- Instead of randomly choosing the public exponent 'e'.
- We always choose 'e' to be 65537 which is a common RSA practice.
- Reasoning:
  - Since e is prime the chance of gcd(e, φ(n)) != 1 is very low if we choose p and q good enough so this satisfies our e condition.
  - e = 65537 is a small exponent with only two 1s in binary (10000000000000001), making exponentiation fast.
  - It's a widely used, industry-standard value

- With the 3 above improvements, our key generation algorithm is now much more efficient and faster.
- I test this out by trying to generate a 16 bits key (same as our previous version) and see that the time taken are much faster and more consistent.

```
src $ python rsa.py
Generating RSA key pair...
Key generation (keysize = 16) took 0.000027 seconds
Public Key (e, n): (65537, 31373)
Private Key (d, n): (22721, 31373)

src $ python rsa.py
Generating RSA key pair...
Key generation (keysize = 16) took 0.000036 seconds
Public Key (e, n): (65537, 45901)
Private Key (d, n): (15041, 45901)

src $ python rsa.py
Generating RSA key pair...
Key generation (keysize = 16) took 0.000055 seconds
Public Key (e, n): (65537, 19781)
Private Key (d, n): (1973, 19781)

src $ python rsa.py
Generating RSA key pair...
Key generation (keysize = 16) took 0.000099 seconds
Public Key (e, n): (65537, 36391)
Private Key (d, n): (21473, 36391)

src $ python rsa.py
Generating RSA key pair...
Key generation (keysize = 16) took 0.000035 seconds
Public Key (e, n): (65537, 34579)
Private Key (d, n): (30473, 34579)

src $ python rsa.py
Generating RSA key pair...
Key generation (keysize = 16) took 0.000033 seconds
Public Key (e, n): (65537, 43039)
Private Key (d, n): (1025, 43039)
```

d) Improving encryption algorithm

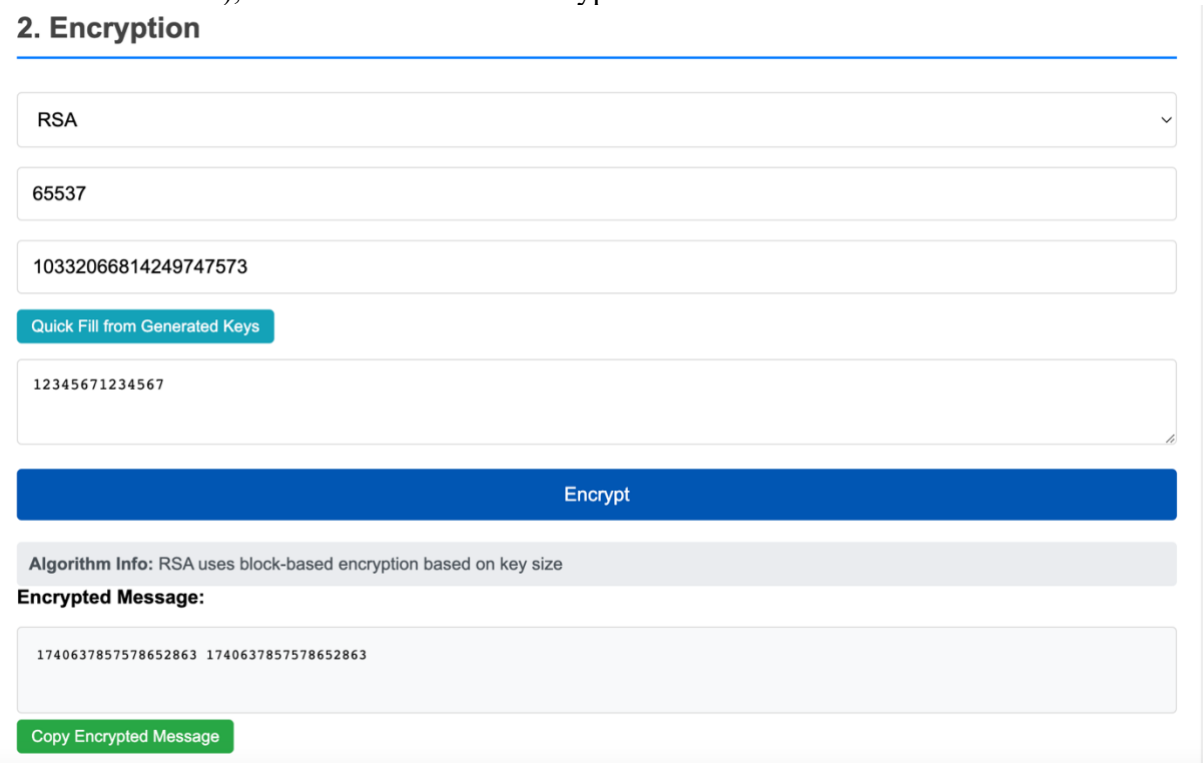
- In our previous version, we encrypt byte by byte. This is not so efficient as RSA can encrypt up to the size of $n - 1$ in one block, we can improve efficiency by encrypting larger blocks instead of individual bytes.

```
        # Split message into blocks and encrypt each
        for i in range(0, len(message_bytes), max_block_size):
            block = message_bytes[i:i + max_block_size]

            # Convert block to integer
            block_int = int.from_bytes(block, byteorder='big')

            # Encrypt block
            encrypted_block = self.encrypt(block_int, public_key)
            encrypted_blocks.append(encrypted_block)
```

- Full code implementation /src/rsa.py

e) Current problem

- Our algorithm is still deterministic (although now it is harder to figure out as we encrypt by block size not character).
- For demo, let choose our key to be a 64 bits key. Now each of our block will have a maximum size of (64 - 1)//8 = 7 bytes.
- So if we craft an input consist of 2 similar 7 bytes (for example 12345671234567), we can see that the 2 encrypted blocks will be the same.



- Real world application of RSA uses padding scheme to make RSA more secure.

3. PKCS#1 v1.5 padding

- PKCS#1 v1.5 padding is the simplest padding scheme to tackle the deterministic weakness of RSA.
- Padding scheme pads the message with some bytes before applying the RSA encryption (C = M^e mod n).
- Padding structure of PKCS#1 v1.5

  `0x00 || BT || PS || 0x00 || D`

  Where:
  - 0x00: a single zero byte which match the start of the padding
  - BT: block type (0x02 for encryption). There are also other block type for signing and private key operation but we will only focus on encryption
  - PS: Padding string (random non-zero bytes)
  - 0x00: another single zero byte which match the end of the padding
  - D: the actual data we are encrypting
- Requirement:
  - Minimum padding length is 8 bytes for encryption
  - Total length must match RSA modulus size in byte
  - One single block of encryption must have at least 12 bytes (11 padding byte and at least 1 byte of message) so the minimum key length of this padding is 12 bytes which is 96 bits.

Code implementation

I. Padding function

```
    def _pkcs1_v15_pad(self, data: bytes, target_length: int, block_type: int = 2)
-> bytes:
        """
        Apply PKCS#1 v1.5 padding to data.

        Args:
            data: The data to pad
            target_length: Target length in bytes (should be key_size // 8)
            block_type: 2 for encryption

        Returns:
            Padded data as bytes
        """
        if len(data) > target_length - 11:
            raise ValueError("Data too long for PKCS#1 v1.5 padding")

        # PKCS#1 v1.5 format: 0x00 || BT || PS || 0x00 || D
        # Where BT is block type, PS is padding string, D is data

        padding_length = target_length - len(data) - 3

        # Encryption operation: padding string is random non-zero bytes
        padding_string = b''
        for _ in range(padding_length):
```

```
            # Generate random non-zero byte
            byte_val = secrets.randbits(8)
            while byte_val == 0:
                byte_val = secrets.randbits(8)
            padding_string += bytes([byte_val])


        # Construct padded message: 0x00 || BT || PS || 0x00 || D
        padded_data = b'\x00' + bytes([block_type]) + padding_string + b'\x00' +
data

        return padded_data
```

- We need 1 zero byte at the start of the padding, 1 zero byte at the end of the padding, and at least 8 bytes of random value. Therefore, the longest length of our data can only be $modulus_{size} - 1 - 1 - 8 = modulus_{size} - 11$.
- We use the `secrets` library to generate cryptographic safe random bits. Remember that our random bits cannot be zero or else when decrypting the message we would not be able to separate our message from the padding.

II. Unpad function

- For unpadding, we simply remove all the bytes from the first zero byte to the next zero byte

```
    def _pkcs1_v15_unpad(self, padded_data: bytes, block_type: int = 2) -> bytes:
        """
        Remove PKCS#1 v1.5 padding from data.

        Args:
            padded_data: The padded data
            block_type: Expected block type (2 for encryption)

        Returns:
            Original data without padding
        """
        if len(padded_data) < 11:
            raise ValueError("Invalid padded data length")

        # Check first byte (should be 0x00)
        if padded_data[0] != 0x00:
            raise ValueError("Invalid PKCS#1 v1.5 padding: first byte not 0x00")

        # Check block type
        if padded_data[1] != block_type:
            raise ValueError(f"Invalid block type: expected {block_type}, got
{padded_data[1]}")

        # Find the 0x00 separator after padding string
        separator_index = -1
        for i in range(2, len(padded_data)):
            if padded_data[i] == 0x00:
                separator_index = i
```

```
                break

        if separator_index == -1:
            raise ValueError("Invalid PKCS#1 v1.5 padding: no separator found")

        # Check minimum padding length (at least 8 bytes for block type 2)
        if block_type == 2 and separator_index < 10:
            raise ValueError("Invalid PKCS#1 v1.5 padding: insufficient padding
length")

        # Extract original data
        original_data = padded_data[separator_index + 1:]

        return original_data
```

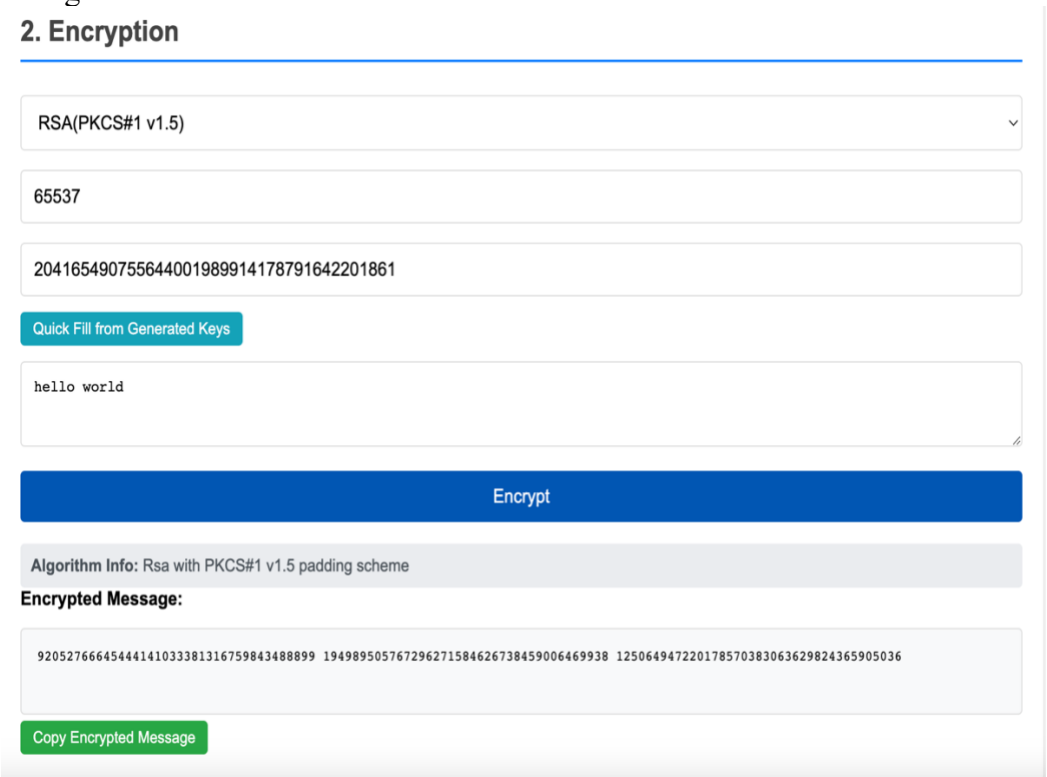
III. Encryption, decryption

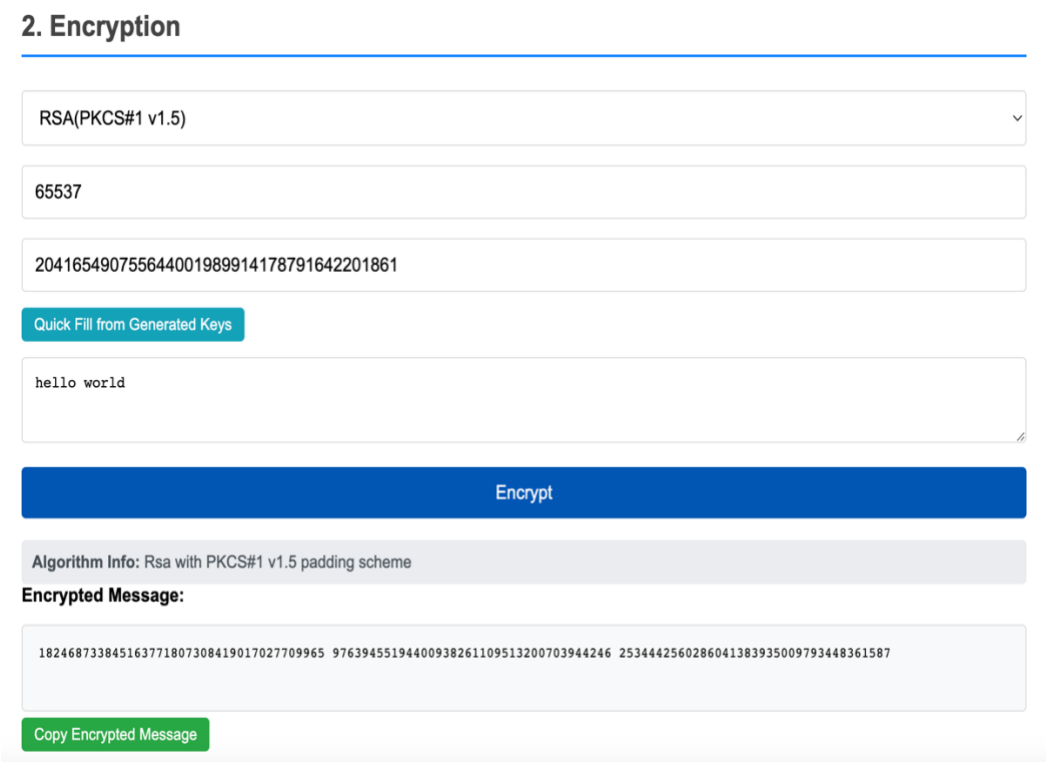
- After apply the padding, we process the encryption as normal.
- For decryption, we first decrypt the message, then remove the padding using the unpad function above.

Full code here: /src/rsa_pkcs.py

Demo

- Now our encryption is no longer deterministic. Encrypting the same text twice now given different result.





Problem

- RSA PKCS#1 v1.5 is now obsolete and can be exploit by chosen cipher text attack (one of these is The Bleichenbacher Attack which will be demo later in this documentation).
- Solution: use RSA OAEP padding (a better padding scheme)

4. RSA OAEP

- Structure:
  `EM = 0x00 || maskedSeed || maskedDB`

  - EM (encode message after padding) must be the same length as the RSA modulus in bytes
  - maskedSeed and maskedDB are generated using a random seed and a mask generation function (MGF1).
- Key requirement:
  `Total overhead = h (seed) + h (lHash) + 1 (0x01) + 1 (0x00) = 2h + 2`

  - We use SHA-256 with has output `h = 32` bytes. Therefore, our minimum key length must be `2*32 + 2 + 1 = 67 bytes = 536 bits.`
- Set up:
  - k: RSA modulus size in bytes
  - mlen: message length in bytes
  - Hash: hash function
  - hLen: output size of hash function (32 bytes for SHA-256)
  - label: optional label
- Encryption process:
  - Hash the label: `lHash = Hash(label)`

- Create data block: `DB = lHash || PS || 0x01 || M`
  `PS` = padding string of 0x00 bytes so that total size match
  `M` = message
  Total length of `DB = k - hLen - 1` bytes
- Generate random seed: `seed = random(hLen bytes)`
- Mask `DB` using the seed
  `dbMask = MGF1(seed, k - hLen - 1)`
  `maskedDB = DB XOR dbMask`
- Mask the seed using the `maskedDB`
  `seedMask = MGF1(maskedDB, hLen)`
  `maskedSeed = seed XOR seedMask`
- Concatenate to form the `EM` (encoded message)
  `EM = 0x00 || maskedSeed || maskedDB`
- Encrypt using RSA algorithm

- Decryption process:
  - Decrypt ciphertext using RSA to get `EM`.
  - Split: `EM = Y || maskedSeed || maskedDB (ensure Y = 0x00)`
  - Unmask:
    `seed = maskedSeed XOR MGF1(maskedDB)`
    `DB = maskedDB XOR MGF1(seed)`
  - Parse `DB`: `DB = lHash || PS || 0x01 || M`
    Check `lHash` matches
    Find `0x01` separator, everything after is your message
- MGF1
  - MGF1 is a deterministic function that expands a short seed into a pseudorandom byte string of any desired length.

    Input:
  - seed
  - mask_len (the length of the mask you want to generate)
  - hash_func (a hash function, we use SHA-256)

    Step:
  - Initialise a counter (`i = 0`)
  - Repeatedly hash the seed concatenated with a 4-byte big-endian counter (i)
  - Concatenate the hash outputs until you've generated `mask_len` bytes.
  - Truncate the result to exactly `mask_len`.

Full code: /src/rsa_oaep.py

How does this improve from PKCS#1 v1.5

- OAEP masks both the message and the random seed in a complex way (using MGF1).
- PKCS#1 v1.5 uses a fixed 0x00 0x02 header and padding string that can be guessed or checked by oracles (this is the key idea for chosen ciphertext attack on

PKCS#1 v1.5). For OAEP, decoding fails silently unless the full hash and structure checks out.

5. **Demoing**

- I have also written some web interface and GUI interface that you can use to play around with RSA. I recommend try using the web interface is it is easier to navigate (the GUI version I stop updating with later RSA version). Follow the instruction after running 'app.py' in my repo to test out RSA algorithm.

D.Breaking RSA

1. **Factoring attack**

- This attack targets the core of RSA assumption of factoring a large composite number is hard (Padding scheme will not help prevent this kind of attack)
- I implement 3 ways we could carry out this attack.
- The code can be found here: factoring rsa

a) Trial division

- In this attack, we just brute force factoring the modulus n into 2 prime p and q.
- We can achieve this by just trying to divide n by number from 2 up to $\sqrt{n}$

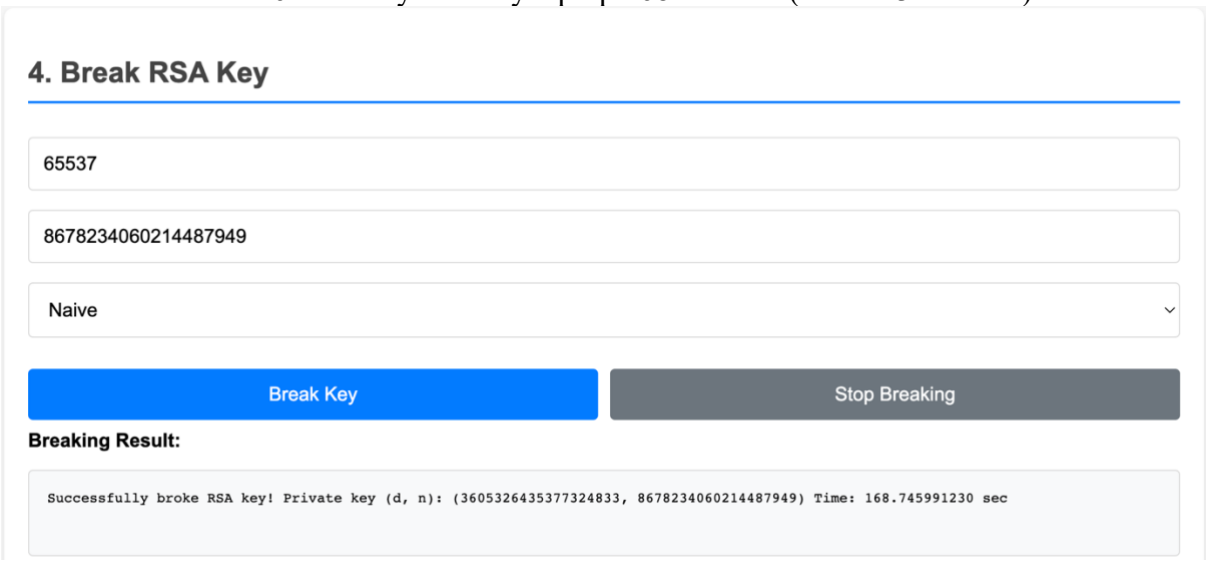
```
    def trial_division(public_key: Tuple[int, int]) -> Tuple[int, int]:
        """
        Naive brute force approach: try to divide n by every prime number up to
sqrt(n).
        Only work for very small n
        """
        e = public_key[0]
        n = public_key[1]
        for i in range(2, int(n**0.5) + 1):
            if n % i == 0:
                p = i
                q = n // i
                phi_n = (p-1)*(q-1)
                d = MathUtils.mod_inverse(e, phi_n)
                return (d,n)
        return (None, None)
```

- Since $n = p \times q$ where p and q are prime numbers, the only factor of n will be p and q. Therefore, whenever we found a factor 'i' of 'n', we essentially success of breaking the RSA encryption.
- However, this brute force method will only work for very small n.
- This approach checks up to $\sqrt{n}$ values, which means for an n bits number, it performs around divisions $2^{n/2}$.
- I use a simple script to see roughly how many divisions per second can my MacBook pro 2022 perform in python.

```
import time

N = 10_000_000
start = time.time()
for i in range(1, N):
    _ = 12345678901234567890 // i
end = time.time()
```

```
print(f"{N:,} divisions in {end - start:.2f} seconds")
print(f"{N / (end - start):,.0f} divisions per second")
```

- It shows that my laptop can do around 24 million divisions per second.
- So it would take me under 5 minutes to break a 64 bits key but in order to break a 128 bits key, I would take me around 24,000 years.
- This is a demo for 64 bits key took my laptop 168 seconds (around 3 minutes)



- You can play around with the web interface to see how large of a key and how long it takes for your computer to break it by brute force.
- Here a sample estimation

Key size (bits)	Time to factor (trial division)
40	4 milliseconds
48	0.7 seconds
56	11 seconds
64	3 minutes
128	24,000 years

b) Pollard's Rho algorithm

- Input: a composite number n, where $n = p \times q$ and p and q are unknown primes
- Set up:
  - Choose a random starting value $x = 2$ (or any $1 < x < n$)
  - Choose a function, typically $f(x) = x^2 + 1 \; mod \; n$
  - Set two variables: x (the slow pointer) and y (the fast pointer) which is initialised to the same value.
- Loop:
  - Update the 2 pointers $x = f(x), y = f(f(y))$
  - Compute $d = \gcd(|x - y|, n)$
  - If d = 1: keep looping
  - if d = n: failure
  - if $1 < d < n$: success, we found a nontrivial factor (either p or q)
- Output: $p = d$ and $q = n//d$

```
    def pollards_rho(public_key):
        """
        Pollard's Rho algorithm to factor n and compute the RSA private key.
        Returns (d, n), where d is the private key exponent.
        """
        e, n = public_key

        def f(x):
            return (x * x + 1) % n

        x, y, d = 2, 2, 1

        while d == 1:
            x = f(x)
            y = f(f(y))
            d = math.gcd(abs(x - y), n)

        if d == n:
            raise Exception("Fails to factor")

        # Successfully factored: n = p * q
        p = d
        q = n // d
        phi_n = (p - 1) * (q - 1)

        d = MathUtils.mod_inverse(e, phi_n)

        return (d, n)
```

Example walkthrough of Pollard Rho algorithm to factor 91

- Set up
  - Define $f(x) = x^2 + 1 \; mod \; 91$
  - Initial values $x = 2, y = 2$
- Step 1:
  Compute:
  $x = f(x) = f(2) = 2^2 + 1 = 5 \; mod \; 91 = 5$
  $y = f\big(f(y)\big) = f\big(f(2)\big) = f(5) = 5^2 + 1 = 26 \; mod \; 91 = 26$
  Compute gcd:
  $\gcd(|x - y|, n) = \gcd(21,91) = 7$
- Step 2:
  The gcd we found is 7 which is nontrivial. Therefore, we can conclude the other factor is $\frac{91}{7} = 13$. We have successfully factor $n = 91$ to 2 primes $p = 7 \; and \; q = 13$.

Why does Pollard Rho work

Suppose:

- p is a factor of n
- We find 2 different x and y such that:

$$x \equiv y \; mod \; p$$

This means that p is a factor of x - y.

- If $x \not\equiv y \bmod n$ then computing gcd $(x - y, n)$ will give us a non trivial factor of n. In our case 'n' has only 2 factor which are p and q, hence we could find p and q

How do we find such x and y?

Rather than randomly guessing pairs (which is inefficient),
Pollard's Rho uses a **pseudo-random sequence** defined by a polynomial function like:

$$f(x) = x^2 + 1 \bmod n$$

This generates a sequence:

$$x_1, x_2 = f(x_1), x_3 = f(x_2)$$

- This sequence is modulo n but it also behaves as modulo p.
- Because there are only finitely many values mod p, the sequence $\{x_k \bmod p\}$ must eventually repeat (similar to the birthday paradox).
  When this repetition happens, we get $x \equiv y \bmod p$ and therefore we can find the prime factor p and q with the step mentioned above

Cycle detection

To detect this repetition without storing the entire sequence, we use **Floyd's cycle detection**. The idea of Floyd's cycle detection is that if we got a cycle and we have 2 pointers, one move slow (the tortoise) and one move fast (the hare), these 2 pointers will eventually meet at the intersection point of the cycle.

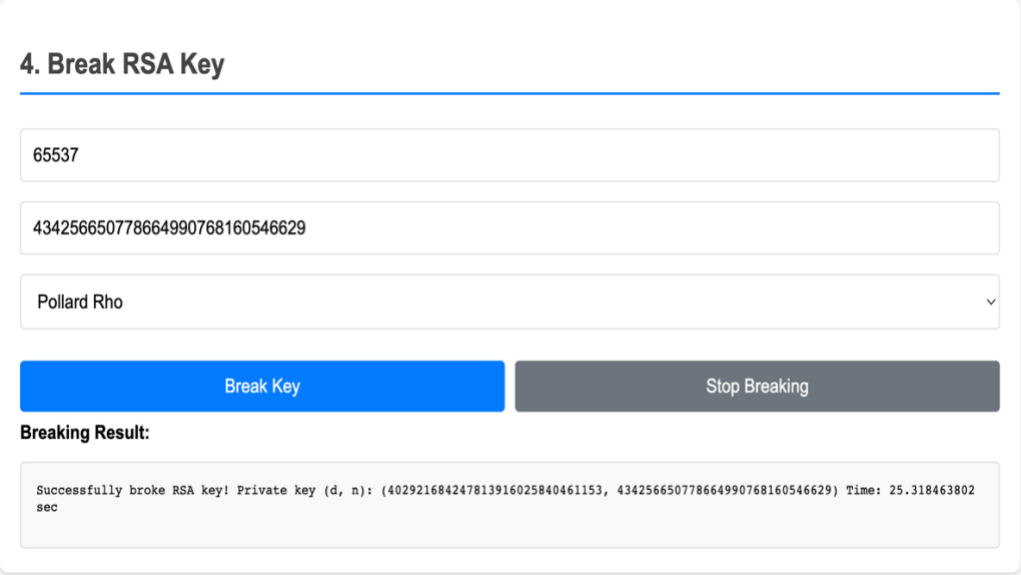
- Let **tortoise** move by 1 step: $x = f(x)$
- Let **hare** move by 2 steps: $y = f(f(y))$
- At each step we check if our 2 pointer meets which is the start of the cycle (the pace where $x \equiv y \bmod p$) by computing gcd $(x - y, n)$. If this gcd is nontrivial, it means we found the cycle (which is what we needed) and the algorithm stops.

Choosing inputs and randomness

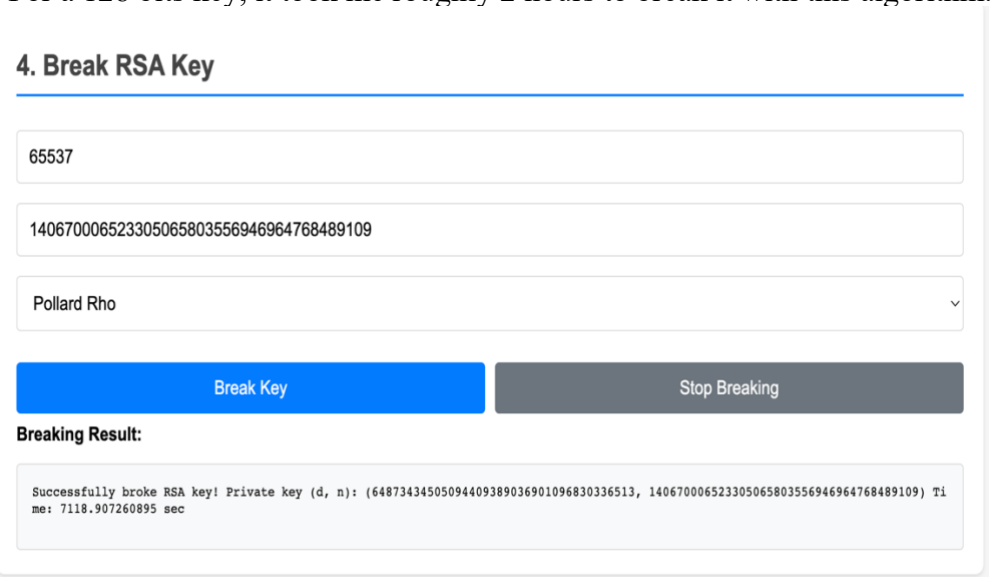
- The starting value x_o and constant c in $f(x) = x^2 + c \bmod n$ are chosen randomly.
- If the algorithm fails (gcd = n), it just restarts with a new function or seed.

Efficiency

- Compared to brute force trial division (which takes $O(\sqrt{n})$ time), Pollard's Rho heuristically runs in $O(n^{1/4})$ times.
- It is especially effective when p or q is small which make our cycle detection faster.
- With my personal laptop, I can easily break a 100 bits key in under 30 seconds



- For a 128 bits key, it took me roughly 2 hours to break it with this algorithm.



- Assuming p and q are chosen well (none of the value are too small), here is an estimation of time it would take to break some of the key length.

Key size (bits)	Time to factor (pollard rho)
100	30 seconds
128	2 hour
256	522,000 years
512	Forever

c) Fermat's Factorisation method

- Base one the representation of an odd integer

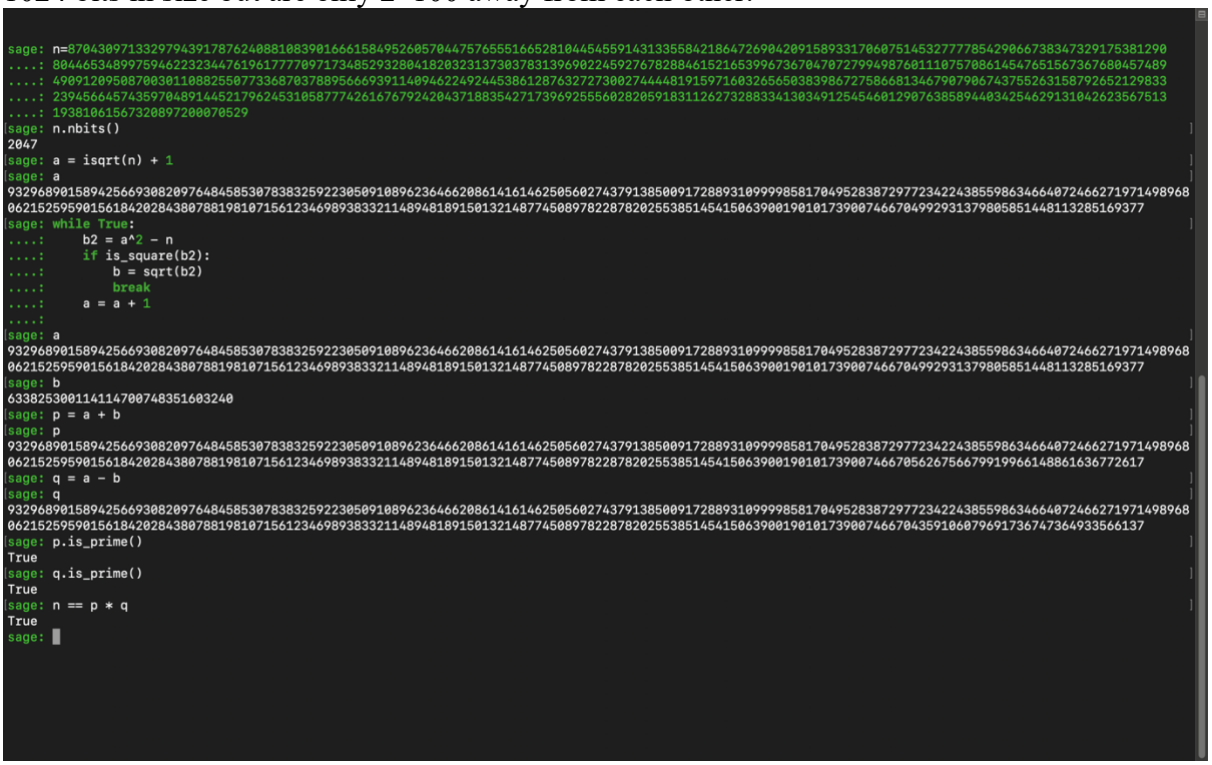
$$N = a^2 - b^2 = (a - b)(a + b)$$

- In our case, we are trying to factor $n = p \times q$. If p and q are close to each other, then a will be close to $\sqrt{n}$, and b will be relatively small.
- The method starts with computing $a = \lceil \sqrt{n} \rceil$ and repeatedly checks whether $a^2 - n$ is a perfect square b^2.

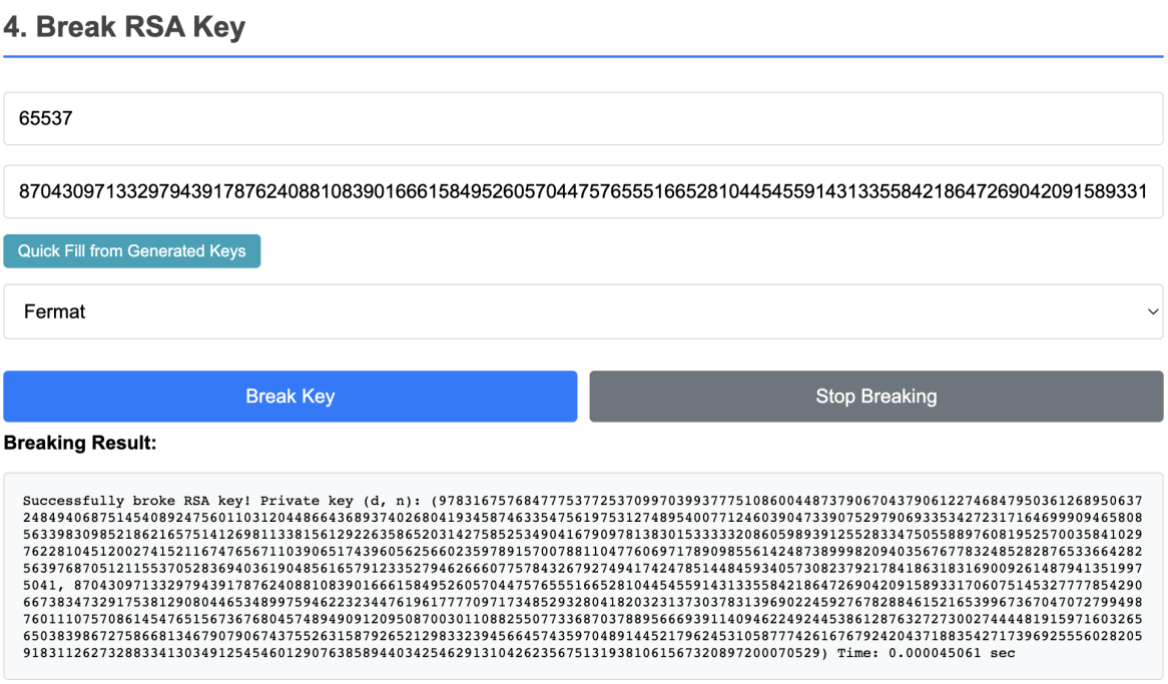
- Once such a and b are found, the factors of n can be computed as $(a - b)$ and $(a + b)$

Efficiency

- Fermat's factorization doesn't depend on the bit length of the key. In fact, it work the best when p and q are relatively close (which is a poorly generated key). This algorithm can break up to a 2048 poorly generated key.
- Here an 2048 bits key that p and q are closed. In these cases, p and q are around 1024 bits in size but are only 2^100 away from each other.



- As you can see, I was able to break a 2048 bits key using my laptop in no time which should not be possible with my computer power (brute forcing this would take millions of years). However, since this key was poorly generated with p and q be very close, Fermat factorisation was able to break it easily (less than even 0.0001 second). You can see this by trying on the web frontend.



Here is the input if you want to try out yourself:

```
N = 870430971332979439178762408810839016661584952605704475765551665281044545591431335584218647269042091589331706075145327777854290667383473291753812908044653489975946223234476196177709717348529328041820323137303783139690224592767828461521653996736704707279949876011107570861454765156736768045748949091209508700301108825507733687037889566693911409462249244538612876327273002744448191597160326565038398672758668134679079067437552631587926521298332394566457435970489144521796245310587774261676792420437188354271739692555602820591831126273288334130349125454601290763858944034254629131042623567513193810615673208972000705 29
```

How to prevent these attacks?

- Use large RSA key (at least 2048 bits, industry standard is now 4096 bits).
- The key should be generated properly:
  - ‘p’ and ‘q’ should not be too close, being vulnerable to Fermat’s Factorisation
  - ‘p’ and ‘q’ too far which is vulnerable to Pollard Rho attacks

2. Chosen ciphertext attack

- RSA encryption has a homomorphic property with respect to multiplication, meaning that the product of two ciphertexts, when decrypted, equals the product of the corresponding plaintexts. This is the core idea when carrying out a chosen plain text attack on RSA

Attack plan

- Let $c = m^e \ mod \ n$ be the cipher text the attacker wants to decrypt
- Step 1: choose a value s with $gcd(s,n) = 1$.
- Step 2: Compute the modified cipher text
  - $c' = c \times s^e \ mod \ n = m^e \times s^e \ mod \ n = (m \times s)^e \ mod \ n$
  - This is the encryption of $m \times s$

- Step 3: Query the decryption oracle with c'
  - If we can decrypt the modified cipher text c' (using a decryption oracle):
$$m' = (c')^d = (m \times s)^{e \times d} = m \times s \; mod \; n$$
- Step 4: Recover the original plain text:
  - Since the attacker knows r, they can compute:
$$m = m' \times s^{-1} mod \; n$$
  - Hence the original plain text is revealed

What is a decryption oracle?

- A decryption oracle is not an intentional feature but rather an unintended side effect of how some cryptographic systems is designed or implemented. Below are key reasons why decryption oracles may exist in real-world scenarios:
  - Error messages and padding checks: many systems return different error responses depending on whether decryption succeeds or fails.
  - For example, PKCS#1 v1.5 padding when trying to decrypt an invalid message (incorrect structure of padding) may return an error like "Decryption error". If the padding is correct, it proceeds to process the plaintext (even if the data is garbage).
  - This allows attackers to distinguish valid vs. invalid ciphertexts, effectively turning the server into a partial decryption oracle.

The Bleichenbacher Attack on RSA PKCS#1 v1.5

- The Bleichenbacher attack (1998) is a famous adaptive chosen ciphertext attack that exploits a padding oracle in RSA PKCS#1 v1.5 decryption.
- In this attack, our oracle is a padding oracle that returns:
  - "Valid" if the decrypted plaintext follows PKCS#1 v1.5 format.
  - Invalid otherwise.
- The oracle can be used to narrow down the value of the decryption of our modified cipher text ($m \times s \; mod \; n$)
- Recall this is the structure of PKCS#1 v1.5:
  `0x00 || 0x02 || PS || 0x00 || D`
- Thus the smallest valid message is $2 \times 2^{8(k-2)}$ (the first byte is zero so does not contribute to the value). The **largest possible message** with 0x02 at the start is slightly less than 0x03, which make the upper bound of $2 \times 2^{8(k-2)}$ with k being the size of the key (each block of encryption) in bytes.
- Therefore, any valid message in PKCS#1 v1.5 lie in the range $[2 \times 2^{8(k-2)}, 3 \times 2^{8(k-2)})$.
- In the Bleichenbacher attack, the oracle can help us narrow down the decryption of our modify ciphertext:
$$2 \times 2^{8(k-2)} \leq m' = m \times s \mod n < 3 \times 2^{8(k-2)}$$
- Each successful s reduces the possible range of m. The attacker iteratively adjusts s and uses the oracle's responses to:
  - Shrink the interval where m must lie
  - Eventually recover m completely.

Demo

- Code can be found at: bleichenbacher attack
- Took us around 1 minutes to decipher a 256 bits key which is significantly better than using factoring method.

```
[+] Found exact plaintext integer: 4074832113430470292408820350282780192909399703643188396077423733664148591
[+] Total oracle queries: 639352
[+] Recovered message: 'hello'

[✓] Attack success: True
[*] Original:  'hello'
[*] Recovered: 'hello'
Took 70.168426 seconds (1.17 minutes)
```

```
[+] Found exact plaintext integer: 5709253679054128165136413883741295242800145292331570521736612933986876158734618846604009962760595036902954218611696056509971879343557366232674033795796617698525865369999360567366530801922416687734370647722607033433519927161775913200406940411012924486680610593197775557218012447311911892323096402120699756
[+] Total oracle queries: 374251
[+] Recovered message: 'lolololol'

[✓] Attack success: True
[*] Original:  'lolololol'
[*] Recovered: 'lolololol'
Took 1566.447466 seconds (26.11 minutes) to break key size of 1024 bits
```

- This attack does not guarantee success and could fails for some of these reasons:
  - r value calculations: The r parameter search can miss valid ranges due to rounding errors
  - Interval intersection failures: When updating intervals, valid intersections might be lost due to integer arithmetic
  - Modular arithmetic edge cases: Operations with very large numbers can cause overflow or precision loss
  - This is just a simple version for demo purpose which is inefficient and might cost timeout
  - So you might have to run the demo for a few times before success.

E. Encrypted chat

- To demonstrate how RSA might be used in a real world application, I create a simple end-to-end encrypted chat

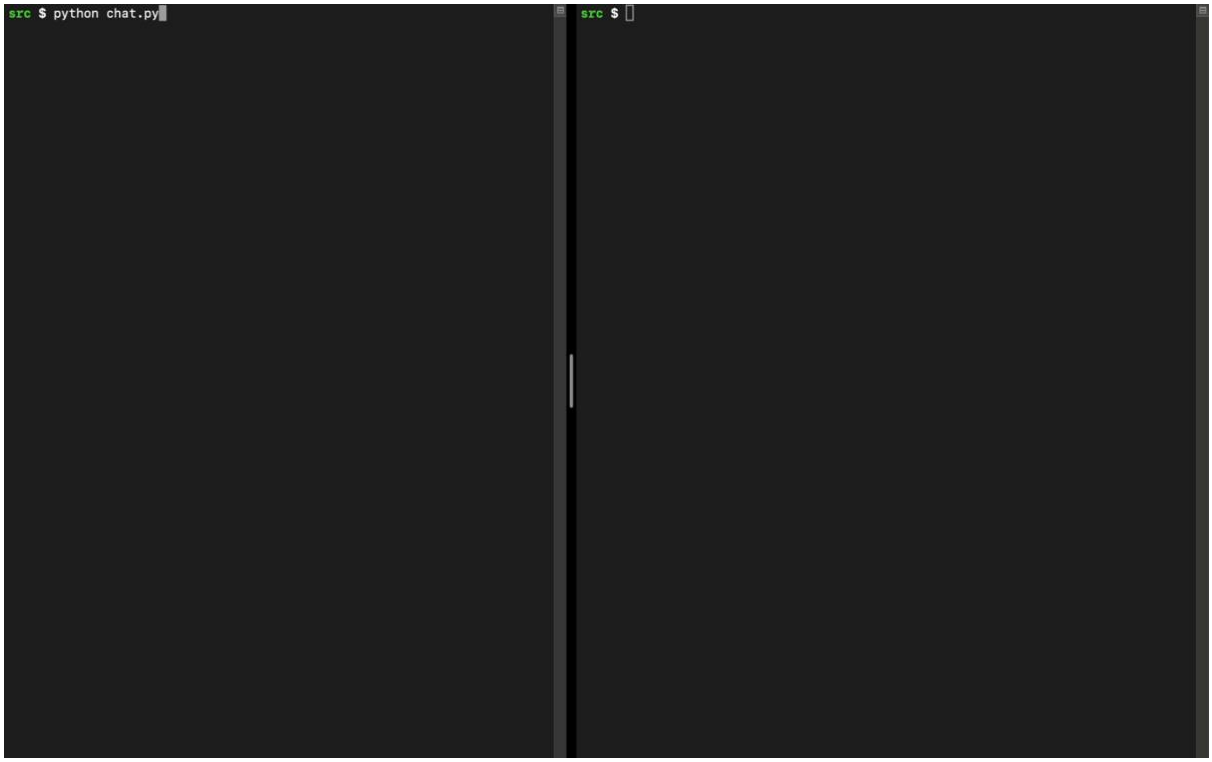
1. Direct chat

- First I build a simple direct chat using a client server architecture over a local network using TCP sockets and multithreading.
- One person will act as a host. The host will provide a known place for the other peer to connect to.
- The other user is the client and can connect a direct channel to the host.

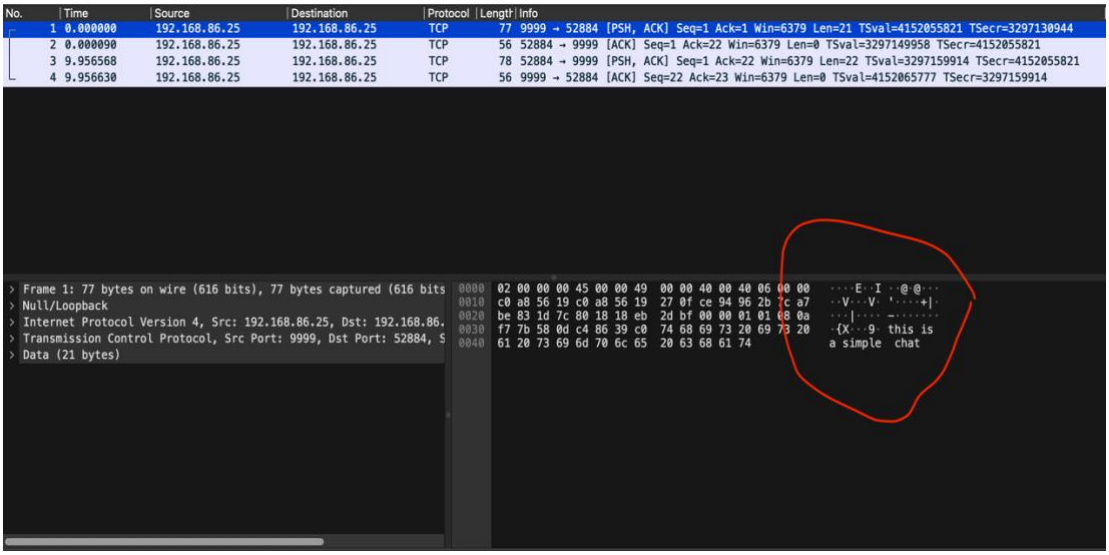
```python
import socket
import threading

choice = input("Do you want to host (1) or to connect (2): ")

if choice == '1':
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("192.168.86.25",9999))
    server.listen()

    client, _ = server.accept()
elif choice == '2':
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(("192.168.86.25",9999))
else:
    exit()

def send_message(c):
    while True:
```

```
        message = input("")
        c.send(message.encode())
        print("You: " + message)


def receive_message(c):
    while True:
        print("Your friend: " + c.recv(1024).decode())


threading.Thread(target=send_message, args=(client, )).start()
threading.Thread(target=receive_message, args=(client, )).start()
```

- For demonstration, I will have 2 instances of the program running on my laptop, one as a host and one as the client connecting. (Double click the play the video).



- Note that there is no encryption yet. An attacker can use a packet sniffing tool like WireShark to sniff and get our message.



2. Adding encryption

- Now we will add RSA to make this chat end-to-end encrypted, meaning that even if an attack sniff and got our message, they would not be able to decrypt it.
- Key exchange process:
  - Host send their key first and they receive the key

```
    # sending the key
    public_key_json = json.dumps(public_key)    # our implemetation of RSA does not
support pem so we sent by json
    client.send(public_key_json.encode())

    # Receiving the key
    data = client.recv(1024).decode()
    partner_key = tuple(json.loads(data))
```

  - Client receives the key first and send their key

```
    # Receiving the key
    data = client.recv(1024).decode()
    partner_key = tuple(json.loads(data))

    # sending the key
    public_key_json = json.dumps(public_key)
    client.send(public_key_json.encode())
```

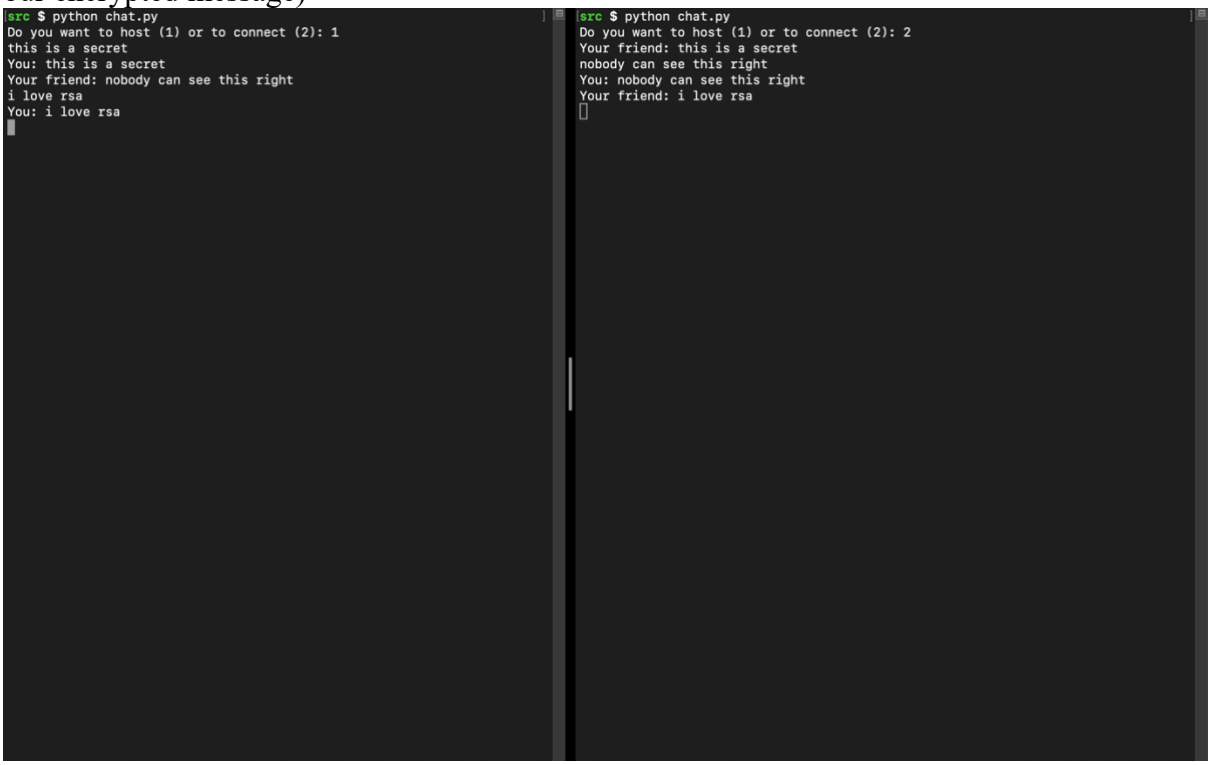
- Encrypt message before sending

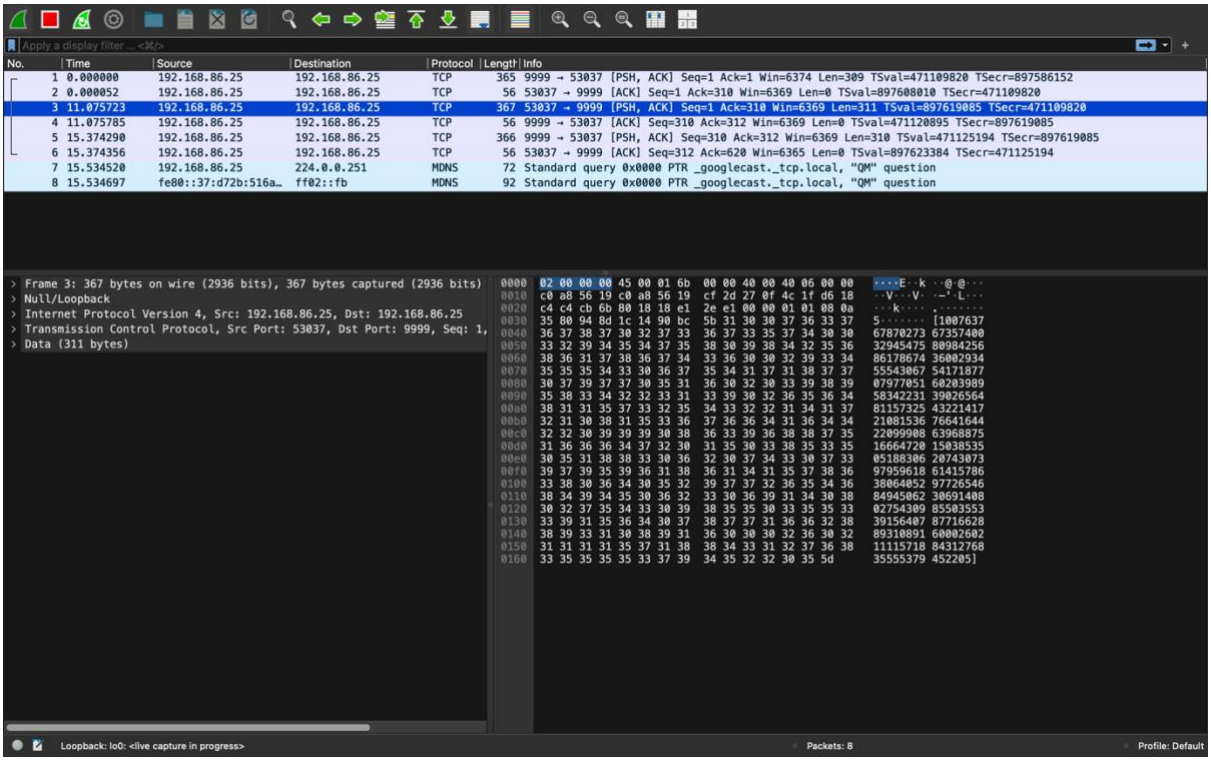
```
        message = input("")

        encrypted_mesage = rsa_instance.encrypt_string(message, partner_key)

        c.send(json.dumps(encrypted_mesage).encode())
```

- Decrypt message after receiving

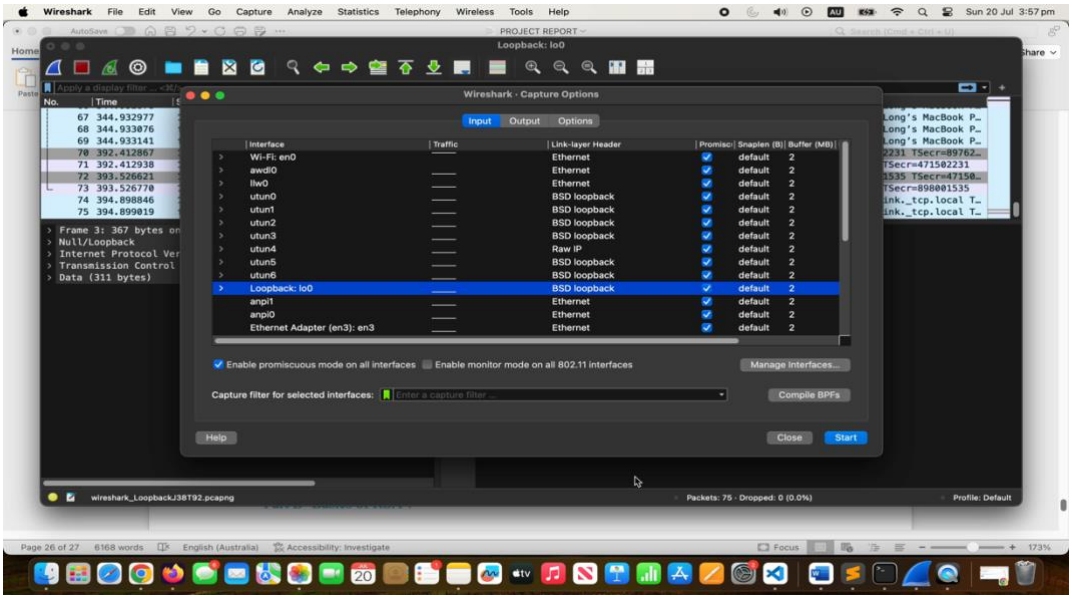
```
        receive = c.recv(1024).decode()
        message = rsa_instance.decrypt_string(json.loads(receive), private_key)
```

- Now when we sniff the chat all we receive is a bunch of number blocks (which is our encrypted message)





- Here is a video demo if you are interested (if you are viewing this file as a pdf file you will need to find the demo in the playlist below):



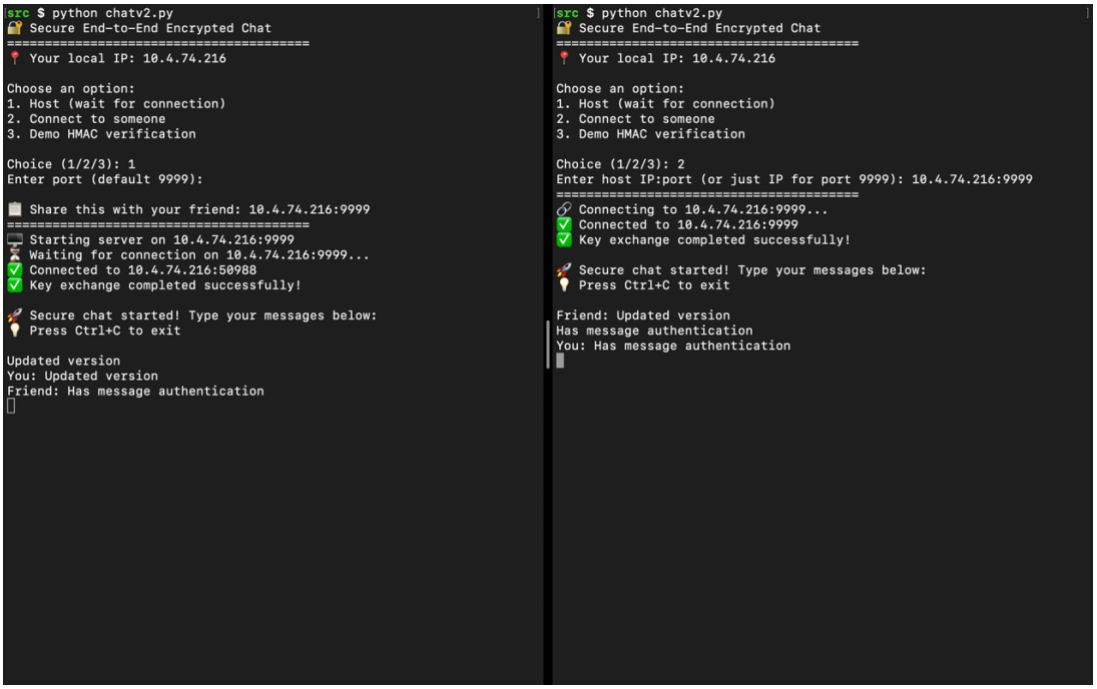
- Limitations:
  - Currently sending keys as raw Python tuples via JSON should use PEM (industry standard)
  - Hard coding the IP address
  - Key exchange process is not secured. A malicious actor can intercept the connection and send their own public key and hence can read all the message even though they are encrypted. I wrote a simple script that demo this (mitm_tamper.py), video demo in the playlist below

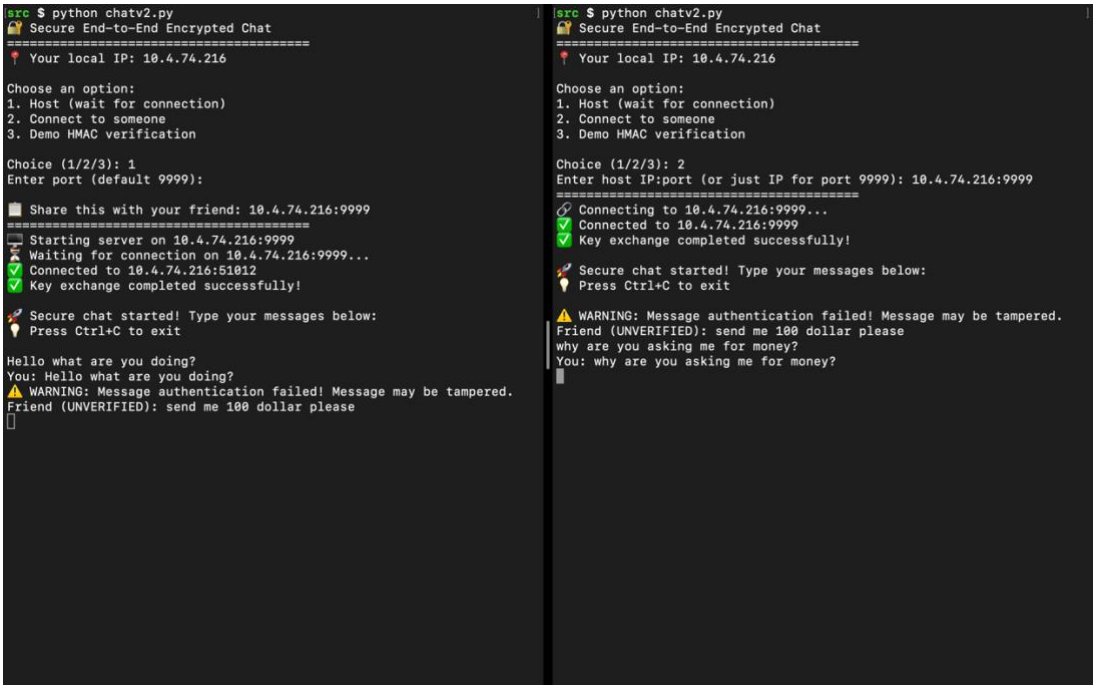
```
============================================================
This demonstrates how MITM attacks work in educational settings
INSTRUCTIONS:
1. First, start one instance of your chat app as HOST (option 1)
2. Then run this MITM script
3. Finally, start another chat app instance as CLIENT (option 2)
   BUT make it connect to port 8888 instead of 9999
4. Watch as messages are intercepted and malicious ones injected
============================================================
[ATTACKER] Proxy server started on port 8888
[ATTACKER] Waiting for connections...
[ATTACKER] User 1 connected from ('10.4.74.216', 51395)
[ATTACKER] Connected to real server at 10.4.74.216:9999
[ATTACKER] Intercepting key exchange...
[ATTACKER] Captured server's public key
[ATTACKER] Sent fake key to user 1
[ATTACKER] Captured user 1's public key
[ATTACKER] Sent fake key to server
[ATTACKER] Key exchange compromised! Both users think they're talking to each other.
[ATTACKER] MITM attack is now active!
[ATTACKER] Press Enter to send malicious messages...
[ATTACKER] Intercepted from Server: 'hello'
[ATTACKER] Forwarded to User 1: 'hello'
[ATTACKER] Intercepted from User 1: 'bye bye'
[ATTACKER] Forwarded to server: 'bye bye'
[ATTACKER] Intercepted from User 1: 'our chat is encrypted so we are safe right?'
[ATTACKER] Forwarded to server: 'our chat is encrypted so we are safe right?'

[ATTACKER] Sent malicious message to User 1: 'I never said that! This is a fake message from the attacker!'

[ATTACKER] Sent malicious message to User 1: 'HACKED! This demonstrates how MITM attacks work!'

[ATTACKER] Sent malicious message to User 1: 'Your account has been compromised - send me your password!'

[ATTACKER] Sent malicious message to User 1: 'Your account has been compromised - send me your password!'
```

  - Can only send limited length message.

Improved version

- I made an improved version of the chat (chatv2.py) with these new features:
  - Automatic IP detection
  - Message authentication using HMAC





- Here a video demo playlist of the above chat: [end-to-end encrypted chat demo]

F. Appendix

Reference list

Part B 'Basics of RSA':

Wikipedia contributors. (2024). *RSA cryptosystem*. Wikipedia. Available at: https://en.wikipedia.org/wiki/RSA_cryptosystem

Part C 'Implementing RSA':

Brilliant contributors. (n.d.). *Extended Euclidean Algorithm*. Brilliant.org. Available at: https://brilliant.org/wiki/extended-euclidean-algorithm/

Crypto Stack Exchange. (2020). *Appropriate public exponent choice for RSA encryption*. Available at: https://crypto.stackexchange.com/questions/76258/appropriate-public-exponent-choice-for-rsa-encryption

Conrad, K., *The Miller–Rabin Test*. Available at: https://kconrad.math.uconn.edu/blurbs/ugradnumthy/millerrabin.pdf

Part D 'Breaking Rsa':

Venkatesh, S. (2014). *Pollard's Rho Algorithm*. University of Colorado Boulder. Available at: https://home.cs.colorado.edu/~srirams/courses/csci2824-spr14/pollardsRho.html

Wikipedia contributors. (2024). *Pollard's rho algorithm*. Wikipedia. Available at: https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm

GeeksforGeeks. (n.d.). *Pollard's Rho Algorithm for Prime Factorization*. Available at: https://www.geeksforgeeks.org/dsa/pollards-rho-algorithm-prime-factorization/

Educative. (n.d.). *Why does Floyd's cycle detection algorithm work?*. Available at: https://www.educative.io/answers/why-does-floyds-cycle-detection-algorithm-work

Asecuritysite. (n.d.). *RSA Example with Calculations*. Available at: https://asecuritysite.com/rsa/rsa_01

YouTube video: https://www.youtube.com/watch?v=-ShwJqAalOk

c0D3M, 2020. *Bleichenbacher attack explained*. Medium. Available at: https://medium.com/@c0D3M/bleichenbacher-attack-explained-bc630f88ff25