Network Security Project - Attack textbook RSA

119033910103 黃竑儒

Introduction

In the project, we attempt to implement textbook RSA algorithm by ourselves. Based on the textbook RSA algorithm we implemented, we try to perform CCA2 attack according to [1] Jefferey et.al,2018. Finally we will try to defend the CCA2 attack by using OAEP key padding algorithm.

Part 1 - Textbook RSA Algorithm

In this part, we will implement textbook RSA algorithm (without any padding). RSA is one of the first public-key cryptosystems and is widely used for secure data transimission. To apply textbook RSA algorithm, we first need to generate RSA key pairs. The **key generation algorithm** is as follows:

1. randomly select two large prime (with specific bit size) p and q, compute $n = p \times q$.
2. compute the Carmichael's totient function $m = \lambda(n) = (p-1) \times (q-1)$.
3. choose an integer e such that $e \in (1, m)$ and $gcd(e, m) = 1$. Usually $e = 65537$ is a common choice.
4. determine d add $d \equiv e^{-1}$ (mod m), that is d is the modular multiplicative inverse of e modulo m.
5. release (n, e) as public key and keep secrect (n, d) as the private key.

There are some worth-mentioning implementation details.

Primality Check for Large Integers

The first step of key generation algorithm is to select two large prime p and q. The *prime number theorem* states that $\frac{n}{\ln n}$ is a good approximation of the number of prime numbers that is less than or equal to n. Therefore, the probability that a randomly chosen number is prime is $\frac{1}{\ln n}$. This theorem gives us a hint that to select two large prime p and q, the most effective way is **firstly sample a large number and check whether it is prime**.

To check whether a randomly sampled integer is prime, one might use the following code:

```python
def is_prime(num: int) -> bool:
    for i in range(2, int(math.sqrt(num))):
        if num % i == 0:
            return False
    return True
```

This is the trivial solution to check prime and its time complexity is O($\sqrt{n}$), which is unacceptable when n is hundreds of bit long. The common key size for RSA is 1024 bit so that both of p and q should be at least 512 bit. Instead, we will use [2] Miller-Rabin algorithm to check whether the chosen number is prime. The running time of this algorithm is O($k \log^3 n$), where n is the number tested for primality and k is the number of rounds performed. We set $k = 128$ in the code.

Extended Eculidean Algorithm

In the 4th step, the modular multiplicative inverse d can be determined using [3] Extended Euclidean Algorithm.

> It follows that both extended Euclidean algorithms are widely used in cryptography. In particular, the computation of the modular multiplicative inverse is an essential step in the derivation of key-pairs in the RSA public-key encryption method.

In short, we can find the solution pair (x, y) of equation $ax + by = gcd(a, b)$ using this algorithm. In the RSA setting, $(a, b) = (e, m)$. Since e and m are coprime, $gcd(e, m) = 1$. We only need $d = x$ so we can discard y. However, usually we will obtain a negative x, that is $x < 0$. But for computation convenience, we want x be positive. Fortunately, $x \equiv x + k \times m$ in modulo and therefore we can obtain a positive $x = d$.

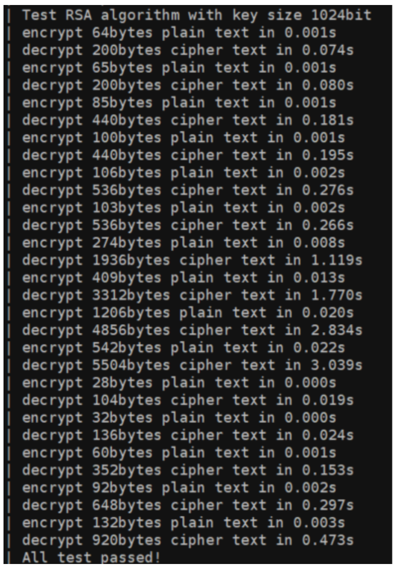
The key generation code is as follows:

```python
def generate_key_pairs(self) -> Tuple[int, int]:
    """

    generate public and private key pairs.

    store the private key and public key.
    only return the public key
    """
    # randomly sample two large prime p and q
    p = utils.sample_prime_with_bit_size(self.size // 2)
    q = utils.sample_prime_with_bit_size(self.size - self.size // 2)
    n = p * q

    # compute the Euler function as m
    m = (p-1) * (q-1)

    # randomly pick an integer e that is relatively prime to m
    e = 65537 if 65537 < m else 11

    # compute integer d s.t. ed - 1 = km
    d, _ = utils.ext_euclid(e, m)

    # keep secret the private key and release the public key
    self.public_key = (n, e)
    self.private_key = (n, d)
    return (n, e)
```

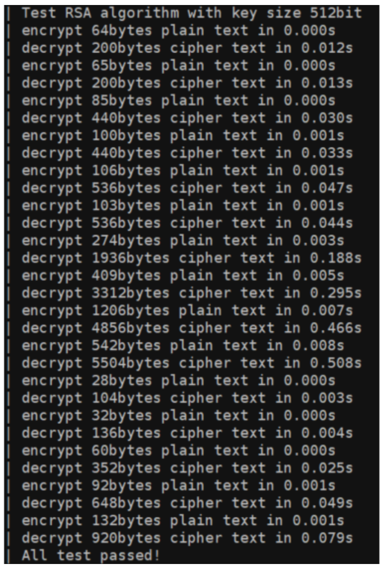
Encryption and Decryption

Now that we have generate the key pair, we can apply RSA algorithm to sensitive data. Without loss of generality, we just consider `str` and `int` data. For any given data, we firstly break it into blocks of data with fixed size 8bit (1 byte). This block data can be guaranteed to be less than n and grant us the ability of dealing with arbitrary large data. The test data we used is stored in *test.txt*.

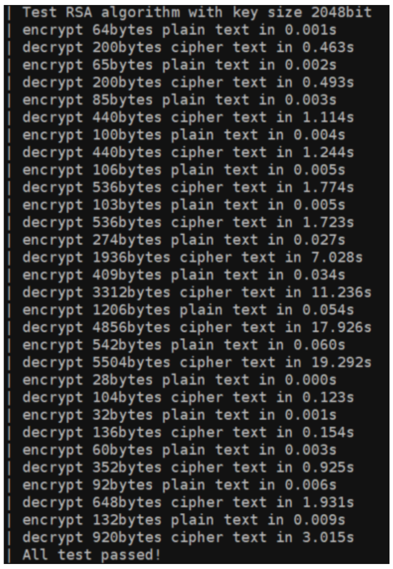
The result of encryption and decryption of different key size is listed as follows:



```
| Test RSA algorithm with key size 1024bit
| encrypt 64bytes plain text in 0.001s
| decrypt 200bytes cipher text in 0.074s
| encrypt 65bytes plain text in 0.001s
| decrypt 200bytes cipher text in 0.080s
| encrypt 85bytes plain text in 0.001s
| decrypt 440bytes cipher text in 0.181s
| encrypt 100bytes plain text in 0.001s
| decrypt 440bytes cipher text in 0.195s
| encrypt 106bytes plain text in 0.002s
| decrypt 536bytes cipher text in 0.276s
| encrypt 103bytes plain text in 0.002s
| decrypt 536bytes cipher text in 0.266s
| encrypt 274bytes plain text in 0.008s
| decrypt 1936bytes cipher text in 1.119s
| encrypt 409bytes plain text in 0.013s
| decrypt 3312bytes cipher text in 1.770s
| encrypt 1206bytes plain text in 0.020s
| decrypt 4856bytes cipher text in 2.834s
| encrypt 542bytes plain text in 0.022s
| decrypt 5504bytes cipher text in 3.039s
| encrypt 28bytes plain text in 0.000s
| decrypt 104bytes cipher text in 0.019s
| encrypt 32bytes plain text in 0.000s
| decrypt 136bytes cipher text in 0.024s
| encrypt 60bytes plain text in 0.001s
| decrypt 352bytes cipher text in 0.153s
| encrypt 92bytes plain text in 0.002s
| decrypt 648bytes cipher text in 0.297s
| encrypt 132bytes plain text in 0.003s
| decrypt 920bytes cipher text in 0.473s
| All test passed!
```



```
| Test RSA algorithm with key size 512bit
| encrypt 64bytes plain text in 0.000s
| decrypt 200bytes cipher text in 0.012s
| encrypt 65bytes plain text in 0.000s
| decrypt 200bytes cipher text in 0.013s
| encrypt 85bytes plain text in 0.000s
| decrypt 440bytes cipher text in 0.030s
| encrypt 100bytes plain text in 0.001s
| decrypt 440bytes cipher text in 0.033s
| encrypt 106bytes plain text in 0.001s
| decrypt 536bytes cipher text in 0.047s
| encrypt 103bytes plain text in 0.001s
| decrypt 536bytes cipher text in 0.044s
| encrypt 274bytes plain text in 0.003s
| decrypt 1936bytes cipher text in 0.188s
| encrypt 409bytes plain text in 0.005s
| decrypt 3312bytes cipher text in 0.295s
| encrypt 1206bytes plain text in 0.007s
| decrypt 4856bytes cipher text in 0.466s
| encrypt 542bytes plain text in 0.008s
| decrypt 5504bytes cipher text in 0.508s
| encrypt 28bytes plain text in 0.000s
| decrypt 104bytes cipher text in 0.003s
| encrypt 32bytes plain text in 0.000s
| decrypt 136bytes cipher text in 0.004s
| encrypt 60bytes plain text in 0.000s
| decrypt 352bytes cipher text in 0.025s
| encrypt 92bytes plain text in 0.001s
| decrypt 648bytes cipher text in 0.049s
| encrypt 132bytes plain text in 0.001s
| decrypt 920bytes cipher text in 0.079s
| All test passed!
```



```
| Test RSA algorithm with key size 2048bit
| encrypt 64bytes plain text in 0.001s
| decrypt 200bytes cipher text in 0.463s
| encrypt 65bytes plain text in 0.002s
| decrypt 200bytes cipher text in 0.493s
| encrypt 85bytes plain text in 0.003s
| decrypt 440bytes cipher text in 1.114s
| encrypt 100bytes plain text in 0.004s
| decrypt 440bytes cipher text in 1.244s
| encrypt 106bytes plain text in 0.005s
| decrypt 536bytes cipher text in 1.774s
| encrypt 103bytes plain text in 0.005s
| decrypt 536bytes cipher text in 1.723s
| encrypt 274bytes plain text in 0.027s
| decrypt 1936bytes cipher text in 7.028s
| encrypt 409bytes plain text in 0.034s
| decrypt 3312bytes cipher text in 11.236s
| encrypt 1206bytes plain text in 0.054s
| decrypt 4856bytes cipher text in 17.926s
| encrypt 542bytes plain text in 0.060s
| decrypt 5504bytes cipher text in 19.292s
| encrypt 28bytes plain text in 0.000s
| decrypt 104bytes cipher text in 0.123s
| encrypt 32bytes plain text in 0.001s
| decrypt 136bytes cipher text in 0.154s
| encrypt 60bytes plain text in 0.003s
| decrypt 352bytes cipher text in 0.925s
| encrypt 92bytes plain text in 0.006s
| decrypt 648bytes cipher text in 1.931s
| encrypt 132bytes plain text in 0.009s
| decrypt 920bytes cipher text in 3.015s
| All test passed!
```

We can see that our self-implemented RSA algorithm work as we expected. One interestng obseveration is that as the key size increases, the time for encryption and decryption also increase. The increment for decryption time is much more significant than the encryption. Because after integer obtained by moduling n is very large and we need to do large power operation when decrypting it. The observation also tells us why we only use RSA to encrypt small size of data, like AES key discussed in the next section, in practice due to its significant time load.

Part 2 - CCA2 attack

In this part we will perform a CCA2 attack on textbook RSA. Textbook RSA is elegant but lack semantic security. An adaptive chosen-ciphertext attack (CCA2) is an interactive form of chosen-ciphertext attack in which an attacker sends a number of ciphertexts to be decrypted, then uses the results of these decryptions to select subsequent ciphertexts.

Client-server Communication Model

Before implementing the CCA2 attack, we firstly build up a client-server communication model. In this communication, the client will

1. generate a 128-bit AES session key for the session
2. encrypt this session key using a 1024-bit RSA public key
3. use the AES session key to encrypt the WUP requset
4. send RSA-encrypted AES session key and the encrypted WUP request to the server

The server will

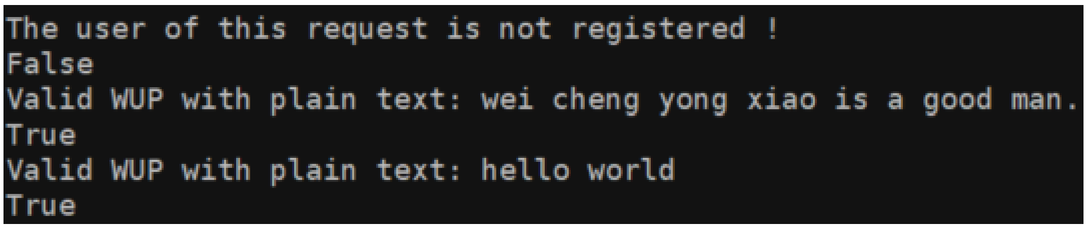
1. decrypt the RSA-encrypted AES key it received from the client
2. choose the least significant 128 bits of the plaintext to be the AES session key
3. decrypt the WUP request using the AES session key
4. send an AES-encrypted response if the WUP request is valid

In our implementation, each client has its own RSA object that maintain the RSA key pairs. The server will register a client and store the client's RSA key pairs before communication.

The original WUP request is somewaht complicated and tedious. We simplify it as the follow formats:

`| content - 1024 bytes | mac - 7 bytes | imei - 7 bytes | SHA - 64 bytes |`

We add 64 bytes SHA at the end of the request. If the check sum is valid, we regard the WUP request as valid. The client-server model test result is as follows:



```
The user of this request is not registered !
False
Valid WUP with plain text: wei cheng yong xiao is a good man.
True
Valid WUP with plain text: hello world
True
```

If the server think the WUP request is valid, it will return true and print out the content of that WUP.

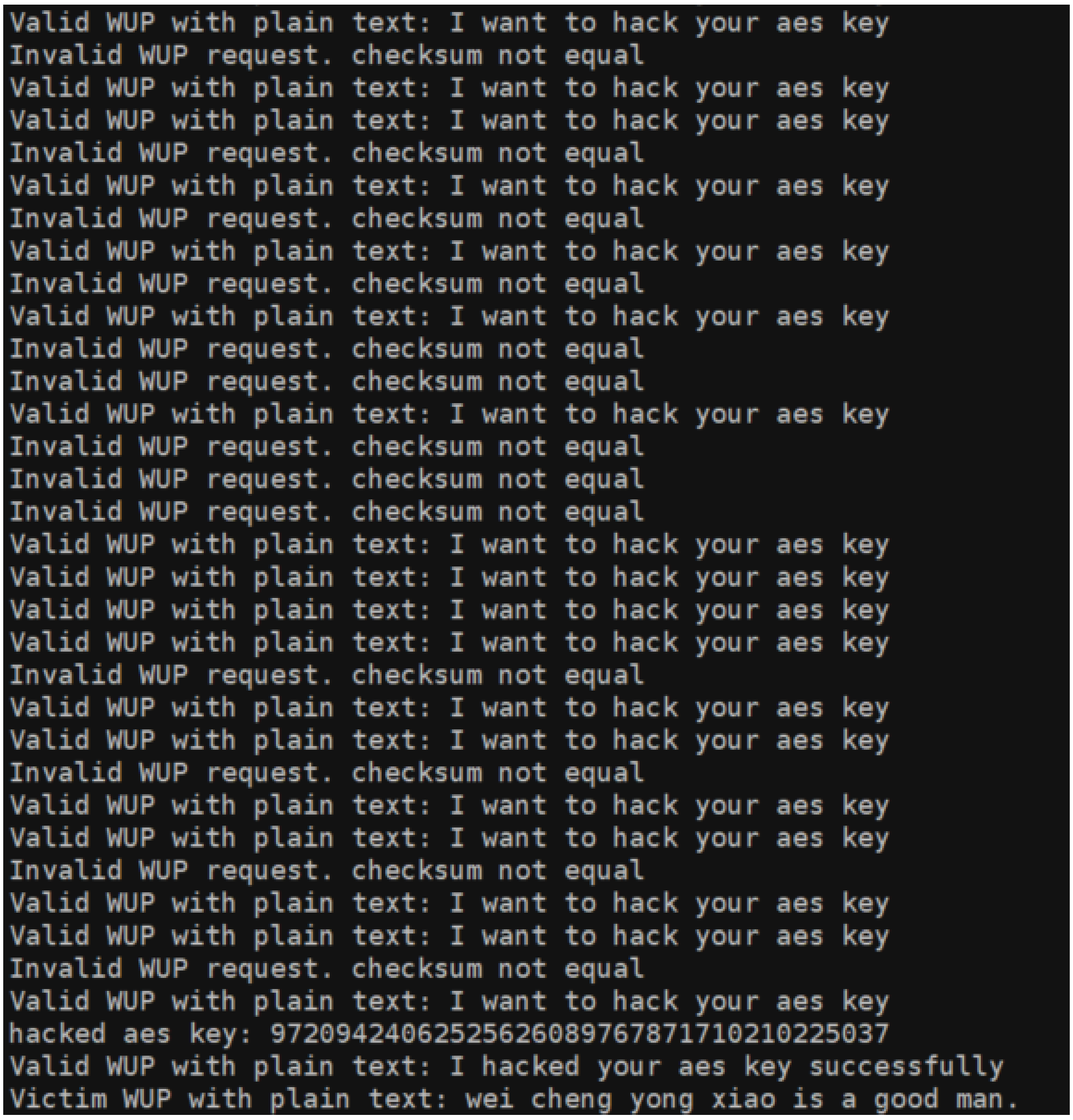
CCA2 attack

The assumption of this attack is that no key padding such as OAEP is used when encrypting the AES key with RSA. Because of this, we are able to leverage the malleability of RSA to perform a chosen ciphertext attack to guess the AES key one bit at a time. Another assumption is that the server will only use the least 128 bit of the encrypted AES key plaintext as the key to decrpyt the message. This is a little bit weird because usually the AES key is 128bit. After it is decrypted correctly, it should be also 128bit. I can't think of any situations that the correctly decrypted AES key is longer than 128 bit except that the situation where we are hacking the key.

The adversary knows RSA public key, a RSA-encrypted AES key and an AES-encrypted WUP request. The adversary guess one bit of the AES key at a time. This part of code is as follows and we give detail comments on the code.

```
        for b in range(127, -1, -1):
            # assume that the encode way is naive encoding
            # according to the fundamental property of multiplication
            # in modular arithmetic, we should compute C_b for each
            # block of this data.
            C_b = [c * ((1 << b*e) % n) % n for c in C]

            # after leftshift, the aes key will be longer than 128 bit.
            # recall that the server only use the least 128bit as the AES key.
            # we mask the least 128bit here.
            aes_key = utils.bit_mask(128) & (k_b << b)
            aes = AES.new(aes_key.to_bytes(16, 'big'))

            # send a dummy WUP request to obtain the server's attitude
            # towards the current hacked aes key.
            wups = WUP("I want to hack your aes key", self.mac, self.imei)
            encrypted_wups = [aes.encrypt(wup) for wup in wups]

            req = [(victim_id, C_b, wup) for wup in encrypted_wups]

            # update the hacked aes key according to server's respondence.
            k_b = (1-server.process_request(req)) << (127-b) | k_b
```

The hacking result is shown as follows:



```
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Valid WUP with plain text: I want to hack your aes key
Valid WUP with plain text: I want to hack your aes key
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
Valid WUP with plain text: I want to hack your aes key
Invalid WUP request. checksum not equal
Valid WUP with plain text: I want to hack your aes key
hacked aes key: 97209424062525626089767871710210225037
Valid WUP with plain text: I hacked your aes key successfully
Victim WUP with plain text: wei cheng yong xiao is a good man.
```

The lines on the top are respondence from the server. We can see that somtimes the dummy WUP request is valid while sometimes it is not. Combine it all we can hack the victim AES key at the bottom line and the further reveal the victim plain text.

Part 3 - OAEP-RSA Algorithm

As we mentioned before, textbook RSA is vulnerable to attacks. One feasible way to defend CCA2 attack in section 2 is to using OAEP key padding algorithm. In cryptography, [4] Optimal Asymmetric Encryption Padding (OAEP) is a padding scheme often used together with RSA encryption.

Recall that in section 1, we cut off the plaintext into blocks with fixed size 8bit. Specifically, for an integer, we cut off the least 8bit of this integer and rightshift 8bit until the integer become 0. For a string, we just cut off it into characters since in python each chracter is 8bit(1 byte) long. We can recover the plaintext by just reversing all of the mention steps.

```
    def naive_encode(self, plain_text: Union[str, int]) -> List[int]:
        if type(plain_text) is str:
            res = [b for b in str.encode(plain_text)]
        else:
            res = []
            mask = bit_mask(8)
            while plain_text:
                res.append(plain_text & mask)
                plain_text >>= 8
        return res

    def naive_decode(self, decode_type: str, plain_text: List[int]) ->
Union[str, int]:
        if decode_type == "str":
            return bytearray(plain_text).decode()
        else:
            pt_ = 0
            for num in plain_text[::-1]:
                pt_ = (pt_ << 8) | num
            return pt_
```

The merit of this preprocessing is in two folds:

- we can handle arbitrary large data theoretically without worrying about the contraint that $m < n$.
- unify different type of plaintext as an integer with fix size, which give convenience for other advanced preprocessing methods, like OAEP.

Now we can implement OAEP following the algorithm described in the reference. The code is as follows:

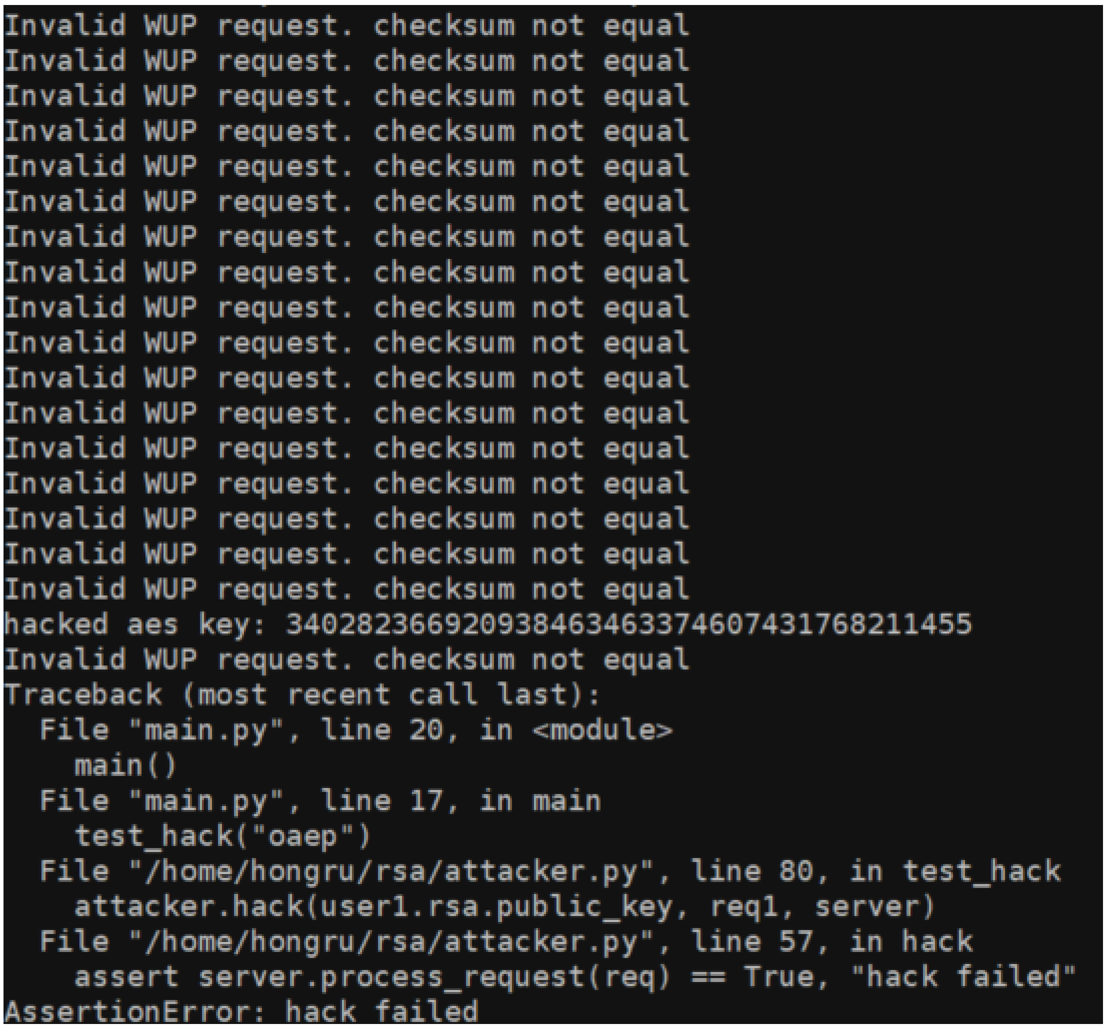
```
    def oaep_encode(self, plain_text: Union[str, int]) -> List[int]:
        """Do OAEP encoding on the basic of naive encoding
        """
        plain_text = self.naive_encode(plain_text)
        oaep_plain_text = []
        for chunk in plain_text:
            m = chunk << self.oaep_k1
            r = randint_with_bit_size(self.oaep_k0)

            X = m ^ self.cryptographic_hash_function(r)
            Y = r ^ self.cryptographic_hash_function(X)

            oaep_plain_text.append((X << self.oaep_k0) | Y)

        return oaep_plain_text

    def oaep_decode(self, decode_type: str, plain_text: List[int]) -> List[int]:
        """OAEP decoding
        """
        oaep_decoded_plain_text = []
        for chunk in plain_text:
            Y = chunk & bit_mask(self.oaep_k0)
            X = chunk >> self.oaep_k0

            r = Y ^ self.cryptographic_hash_function(X)
            m = (X ^ self.cryptographic_hash_function(r)) >> self.oaep_k1

            oaep_decoded_plain_text.append(m)

        return self.naive_decode(decode_type, oaep_decoded_plain_text)
```

We can confirm that the OAEP encoding work along with the RSA algorithm. OAEP can guarantee two goals:

- add an element of randomness (the random sample integer r in the above code) which can be used to convert a deterministic encryption scheme into a probabilistic scheme.
- prevent partial decryption of ciphertexts by ensuring that an adversary cannot recover any portion of the plaintext without being able to invert the trapdoor one-way permutation.

In the CCA2 attack, the adversary only knows the RSA public key, an RSA-encrypted AES key and an AES-encrypted WUP request. However, to fully recover the OAEP-RSA encrypted AES key, it need to know k_0 and k_1 so that it can recover X and Y. The adversary also need to know the hash function used in OAEP setting to further recover m and r. In my opinion, the OAEP can be regarded as another encryption layer before RSA encryption. The CCA2 attack we discussed above only attacks the RSA layer so it can do nothing to the OAEP part, which in turn guarantee the AES key is safe.

We can see that all of the attempted dummy WUP request to hack a OAEP-RSA encrypted AES key is invalid and the final hacking result is failure.



```
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
Invalid WUP request. checksum not equal
hacked aes key: 340282366920938463463374607431768211455
Invalid WUP request. checksum not equal
Traceback (most recent call last):
  File "main.py", line 20, in <module>
    main()
  File "main.py", line 17, in main
    test_hack("oaep")
  File "/home/hongru/rsa/attacker.py", line 80, in test_hack
    attacker.hack(user1.rsa.public_key, req1, server)
  File "/home/hongru/rsa/attacker.py", line 57, in hack
    assert server.process_request(req) == True, "hack failed"
AssertionError: hack failed
```

Conclusion

In this project, we implemented textbook RSA algorithm by ourselves. We perform a CCA2 attack on the implemented textbook RSA algorithm to recognize the semantic vulnerability . At the last part we implement OAEP-RSA algorithm to defend the mentioned CCA2 attack.

Reference

[1] Jeffery Knockel, Thomas Ristenpart, Jedidiah Crandall. 2018. *When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users*

[2] Miller-Rabin algorithm

[3] Extended Euclidean Algorithm

[4] Optimal Asymmetric Encryption Padding