

Multiple Experiments on Textbook RSA

Course Project of IS309 Network Security

薛春宇 518021910698

1. Introduction

RSA 算法 [1] 又称 *plain RSA* 或 *textbook RSA*, 是确定性公钥加密算法的一种, 由 *Ron Rivest*、*Adi Shamir* 和 *Len Adleman* (美国麻省理工学院) 于 1977 年提出, 是目前最有影响力的公钥加密方案之一。由于算法中需要调用 *GenModulus* 生成由两个素数之积构成的大整数, RSA 算法的可靠性依赖于数学中的大整数分解假设, 即对任何类型的攻击者, 在多项式时间内对大整数进行有效的因式分解是困难的。在 3.1 节中, 本项目将对上述 *textbook RSA* 进行实现, 并使用该算法分别完成消息的加密和解密过程。

然而, 虽然 *textbook RSA* 的算法设计的极为巧妙和优雅, 但在语义安全性 *semantic security* 上却有所不足, *Knockel J* 等人于 2018 年在论文 [2] 中给出了一种自适应选择密文的攻击方案, 证明传统的 *textbook RSA* 并不满足 CCA2 安全。在该攻击方案中, 攻击者具有访问解码机的 *oracle* 权限, 可向其发送多条密文并获得相应解密后的明文, 再根据结果选择后续的密文。论文证明该攻击方案能够在有效时间内攻破使用 *textbook RSA* 加密的 AES 会话密钥, 并以此进行数据解密。在 3.2 节中, 本项目参考论文 [2], 实现上述基于 CCA2 的 *textbook RSA* 攻击方案。

上述 *textbook RSA* 的安全漏洞可以通过最优非对称加密填充 (*Optimal Asymmetric Encryption Padding*) 的改进方案进行修复。*OAEP* 算法由 *Bellare* 和 *Rogaway* 在 1994 年首先提出 [3], 是一种随机化的消息填充技术, 而且是从消息空间到一个陷门单向置换 (*One-way Trapdoor Permutation*) 定义域的易于求逆的变换 [4]。在公钥密码学中, *OAEP* 是一种经常与 *RSA* 加密一起使用的填充方案。*OAEP* 对现有 *RSA* 加密方案主要由两大改进:

- 添加可用于将确定性加密方案转换为概率性加密方案的随机性元素;
- 通过确保攻击者不能在无法反转陷门单向置换的情况下恢复明文的任何部分, 来防止密文的部分解密或其他信息的泄漏。

基于上述改进策略, *OAEP* 密钥填充后的 *RSA* 加密方案能够通过在每次加密过程中添加随机元素, 来抵御上文提到的自适应选择密文攻击, 进而保证 CCA2 要求下的加密安全。本项目将参考论文 [3], 实现上述最优非对称加密填充方案, 并将 *OAEP* 应用到 *textbook RSA* 及相应的 CCA2 攻击环境下, 以验证方案的正确性。

2. 环境配置

本项目中的实验环境基于 *Conda Python* 进行搭建, 具体的环境及密码工具包版本如下:

- *Python Version*: 3.6.13
 - *Crypto Version*: 1.4.1
 - *Cryptography*: 3.4.7
 - *Pycryptodome*: 3.10.1
-

3. 实验内容

本节中分别介绍 *textbook RSA*, 针对 *textbook RSA* 的 CCA2 攻击, 以及提高 *textbook RSA* 方案安全性的 OAEP 密钥填充策略的实现方法。

3.1 Textbook RSA 的实现和应用

3.1.1 实现原理

RSA 加密策略依赖于大整数分解的困难性, 首先调用 $\text{GenRSA}(1^\lambda)$ 函数获得构造密钥的 (N, e, d) 。在该函数中, 首先调用 $\text{GenModulus}(1^\lambda)$ 生成两个素数 p, q 以及素数相乘得到的大整数 N 。接着, 由欧拉 ϕ 函数的性质, 可以计算 $\phi(N) = (p - 1)(q - 1)$ 。 GenRSA 算法选择 e 使得 $\text{gcd}(e, \phi(N)) = 1$, 并计算 e 的逆元 d 使得 $ed = 1 \pmod{\phi(N)}$, 返回 (N, e, d) 。*RSA* 加密策略实现的语法如下所示:

- $\text{KeyGen}(1^\lambda) \rightarrow (pk, sk)$: 输入 1^λ , 算法运行 $\text{GenRSA}(1^\lambda)$ 算法获得 (N, e, d) , 输出公钥 $pk := (N, e)$ 和私钥 $sk := (N, d)$
- $\text{Enc}(pk, m) \rightarrow c$: 输入公钥 $pk = (N, e)$ 和明文 $m \in Z_N^*$, 输出密文 $c := m^e \pmod{N}$
- $\text{Dec}(sk, c) \rightarrow m$: 输入私钥 $sk = (N, d)$ 和密文 $c \in Z_N^*$, 输出明文 $m := c^d \pmod{N}$

该方案的正确性如下: 对任何 $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$, 任意 $m \in Z_N^*$, 我们有:

$$\text{Dec}(sk, \text{Enc}(pk, m)) = \text{Dec}(sk, m^e \pmod{N}) = (m^e \pmod{N})^d \pmod{N} = m^{ed \pmod{\phi(N)}} \pmod{N} = m \pmod{N}$$

3.1.2 代码分析

素数生成

首先, 为了实现大整数 (e.g. 1024 bits) 的生成, 需要基于 *Miller Rabin* 素数检测算法, 实现素数生成的相关函数。由 *Bertrand's postulate* 推论可知, 对任何 $n > 1$, n -bit 整数是素数的概率至少为 $1 / 3n$ 。在素数生成算法中, 令 $t = 3n^2$, 则函数返回 *failure* 为可忽略概率: $(1 - 1/3n)^t = ((1 - 1/3n)^{3n})^n \leq (e^{-1})^n = e^{-n}$ 。因此, 实现中的 *while* 循环最多执行 $3n^2$ 即可成功找到素数, 其中 n 是生成素数的 bit 长度。

```
1 # Generate prime with the length of bits equals to key_size
2 def generate_prime(key_size):
3     # print("----- Begin Generating Prime -----")
4     # This loop will end in 3 * pow(length, 2) at most
5     while True:
6         # Set 1 in the highest bit
7         target = random.randrange(2 ** (key_size - 1), 2 ** key_size)
8         if miller_rabin_test(target, key_size):
9             return target
```

RSA 密钥构造

在密钥构造函数 $\text{generate_keys(key_size)}$ 中, 首先分别调用两次素数生成函数得到 p 和 q , 相乘得到大整数 N 。为保持 N 的 bit 长度与输入的 key_size 保持一致, p 和 q 的长度设置为密钥长度的一半, 并多次循环尝试直到满足 N 的长度要求。

```
1 # generate_keys() 函数中实现
2
```

```

3 # Generate big primes
4 print("Generating big primes (may take some time) ...")
5 N = 1
6 prime_p, prime_q = 1, 1
7 # Generate N with the target size
8 while N.bit_length() != key_size:
9     print("Try %d for the generation of big primes..." % counter)
10    # Generate big prime p and q to form N
11    unit_key_size = key_size // 2
12    prime_p = generate_prime(unit_key_size)
13    prime_q = generate_prime(key_size - unit_key_size)
14    N = prime_p * prime_q
15    counter += 1

```

获得大整数 N 后，需要使用欧拉 ϕ 函数的相关性质以及扩展欧几里得算法，计算 $\phi(N)$ ，并以此计算 RSA 密钥中的 e 和 d 。

```

1 # generate_keys()函数中实现
2
3 # Euler phi function of N
4 print("Calculating Euler phi function of N...")
5 phi_N = (prime_p - 1) * (prime_q - 1)
6 # Public key elm, smaller should be better
7 print("Formulating public key and private key...")
8 e = random.randrange(3, phi_N)
9 gcd_res = 0
10 while True:
11     gcd_res, _, _ = extended_euclidean(e, phi_N)
12     if gcd_res != 1:
13         e = random.randrange(3, phi_N)
14     else:
15         break
16 # Private key elm, use extended Extended Euclidean Algorithm
17 gcd_res, d, _ = extended_euclidean(e, phi_N)

```

最后，我们构造 *textbook RSA* 加密方案的公、私钥并返回，至此，RSA 算法中的密钥生成步骤已经完成。将生成的公钥和私钥保存为 `txt` 格式，以供后续加密及解密操作的模块调用。

```

1 # generate_keys()函数中实现
2
3 # Form the keys
4 public_key = (N, e)
5 private_key = (N, d % phi_N)
6 print("Keys generation is completed! | Public key: (N, %d) | Private key: (N, %d)" %
(e, d % phi_N))
7
8 # Write into txt
9 txt_writer(public_key, "./keys/public_key.txt")
10 txt_writer(private_key, "./keys/private_key.txt")

```

RSA 加密

该加密方案的输入为 `txt` 格式的多行文本，位于 `./test/ciphertext.txt` 目录下。我们需要对输入进行逐行处理，包括将文本从字符串格式通过 `utf-8` 编码转为 `bytes` 格式，再转为十进制的 `int`，进行基于公钥的模指数运算后得到同样是十进制形式的密文。完成后，将加密过后的密文写入 `./test/ciphertext.txt` 中，以供后续的解码操作使用。

```
1 # Encrypt
2 def encrypt_plaintext(public_key, file_path, output_path, key_size,
3     key_padding_option, decrypted_OAEP_option):
4     print("----- Begin Encrypting plaintext -----")
5
6     # Ciphertext list
7     ciphertext_list = []
8     # Str list from txt
9     str_list = txt_reader(file_path)
10    for i in range(len(str_list)):
11        cur_str = str_list[i]
12        cur_str = bytes(cur_str, encoding='utf-8')
13        plaintext = int(b2a_hex(cur_str), 16)
14        # 使用基于公钥的模指数运算进行加密
15        ciphertext = fast_exp_mod(plaintext, public_key[1], public_key[0])
16        ciphertext_list.append(ciphertext)
17    # Write to txt
18    txt_writer(ciphertext_list, output_path)
```

RSA 解密

解密模块的输入为已被加密的密文文本，位于 `./test/ciphertext.txt` 目录下。我们同样需要逐行处理输入，将文本从字符串格式转为十进制的 `int` 格式，再通过基于私钥的模指数运算后得到十进制的明文。之后，需要使用 `binascii` 工具包提供的 `a2b_hex()` 函数将明文转为 `bytes` 格式，最后通过 `str` 函数以 `utf-8` 编码标准换为明文字符串。完成后，将解密得到的明文写入 `./test/ciphertext.txt` 中，以验证本 *textbook RSA* 模块的正确性，实验结果将在第 4 节中给出。

```
1 # Decrypt
2 def decrypt_ciphertext(private_key, file_path, output_path):
3     print("----- Begin Decrypting plaintext -----")
4
5     # Plaintext list
6     plaintext_list = []
7     # Str list from txt
8     str_list = txt_reader(file_path)
9     for i in range(len(str_list)):
10        ciphertext = int(str_list[i])
11        # 使用基于密钥的模指数运算进行解密
12        interpret = fast_exp_mod(ciphertext, private_key[1], private_key[0])
13        # Bytes type
14        plaintext = a2b_hex(hex(interpret)[2:])
15        plaintext = str(plaintext, encoding='utf-8')
16        plaintext_list.append(plaintext)
```

```
17     # Write to txt
18     print("Decrypted result:", plaintext_list)
19     txt_writer(plaintext_list, output_path)
```

3.1.3 模块使用方式

在虚拟环境中进入项目文件夹，通过如下命令分别进行 *textbook RSA* 的密钥生成、文件加密和解密。

- 生成指定 *size* 的密钥：

```
1 | python textbook_RSA.py --generate_keys --key_size 1024
```

- 加密指定路径的 *plaintext* 文件（`txt` 格式）：

```
1 | python textbook_RSA.py --encrypt_file ./test/plaintext.txt
```

- 解密指定路径的 *ciphertext* 文件（`txt` 格式）：

```
1 | python textbook_RSA.py --decrypt_file ./test/ciphertext.txt
```

3.2 针对 *textbook RSA* 的 CCA2 攻击

3.2.1 实现原理

交互模型及报文设计

为了进行 CCA2 攻击的模拟实验，需要首先构建合理的 *client - server* 交互模型，并设计 *WUP* 交互报文的格式。对此，本项目进行了如下的交互模型设计：

- *Client* 端：
 - 生成 128-bit 的 AES 密钥作为会话密钥；
 - 使用 1024-bit 的 RSA 公钥来加密会话密钥；
 - 使用 AES 会话密钥对 *WUP* 交互报文的 *request* 进行加密，并添加 *bytes_length* 信息和 *crc* 校验位；
 - 将经过 RSA 加密后的会话密钥和经过 AES 加密后的 *WUP* 交互报文发送给 *server*。
- *Server* 端：
 - 使用 RSA 私钥解密接收到的 AES 会话密钥；
 - 选择上述解密结果中有效的 128-bit 低位作为 AES 会话密钥；
 - 使用 AES 会话密钥和 *bytes_length* 信息解密接收到的 *WUP* 交互报文，并使用 *crc* 校验位判断字段是否合法；
 - 若 *WUP* 交互报文中的 *request* 合法，返回一个 AES 解密成功的响应。

在上述过程中，*WUP* 交互报文的格式包括经过 **RSA** 加密后的会话密钥，**bytes_length** 字段，经过 **AES** 加密后的 **request** 字段和 **response** 字段，以及 **crc** 校验位。将 *WUP* 设计成上述格式的原因主要有四点：

- 考虑 *client* 和 *server* 建立连接及通信的过程，可能存在的交互字段格式包括会话密钥、*request* 和 *response*；
- 将 *WUP* 交互报文设计成包含所有可能传输字段的格式，可以尽量减少特殊报文的单独讨论（例如建立连接阶段的密钥交换），增加 *WUP* 报文的通用性和泛化性；
- 在通信过程中，一方可能会在接收到请求后，首先返回一个 *response*，再发送一个自己提出的 *request*，此时上

述两个字段可以置于同一个 WUP 交互报文中发送，以提高通信的传输效率。若仅需使用 WUP 中的一个字段，则可以直接将剩余的字段设置为空；

- crc 校验位能够保证报文在传输过程中的完整性。根本思想就是先在要发送的帧后面附加一个数，生成一个新帧发送给接收端。到达接收端后，再把接收到的新帧除以这个选定的除数。因为在发送端发送数据帧之前就已通过附加一个数，做了“去余”处理，所以结果应该是没有余数。如果有余数，则表明该帧在传输过程中出现了差错。

在报文交互的过程中，*client* 和 *server* 可以根据当前所需传输的交互信息类别，自主设置 WUP 交互报文的字段，进行高效的相互通信。

CCA2 的攻击原理

本项目参考论文 [2]，利用 *textbook RSA* 的可延展性来完成自适应选择消息攻击。在该攻击方案中，攻击者会首先记录 *client* 和 *server* 之间的会话信息，接着与服务器端建立会话连接，并尝试使用一系列经过转换的 RSA 密文与服务器进行加密通信，以获取有关被攻击客户端使用的原始密钥的信息。获取上述信息后，攻击者只需要使用破获的 AES 密钥对记录的 WUP 交互报文中的 *request* 和 *response* 字段，即可获取交互数据。

令 C 表示使用 RSA 公钥 (N, e) 加密后 128-bit 的 AES 密钥 k ，则有：

$$C \equiv k^e \pmod{N}$$

现在令 C_b 表示 RSA 加密后的 AES 密钥， $k_b = 2^b k$ 表示 AES 密钥 k 向左进行了 b bit 的移位，则有：

$$C_b \equiv k_b^e \pmod{N}$$

我们可以仅通过 C 和公钥来计算 C_b ：

$$C_b \equiv C(2^{be} \pmod{N}) \pmod{N} \equiv (k^e \pmod{N})(2^{be} \pmod{N}) \pmod{N} \equiv (2^b k)^e \pmod{N} \equiv k_b^e \pmod{N}$$

为了开展 CCA2 攻击，我们首先考虑 k_{127} 的 RSA 密文 C_{127} ，其中 k_{127} 是除最高位外的其他位全部为 0 的 AES 密钥，且其最高位是 k 的最低位，原因是在本项目的交互模型中，仅将低位 128-bit 作为 AES 密钥。我们首先猜测 k_{127} 的最高位 bit 是 0，并发送一条 WUP 报文信息和 C_{127} ，并使用猜测的 k_{127} 来加密报文中的 *request* 和 *response* 字段。若能 *server* 正常响应，则说明最高位 bit 确实为 0；若不能，则说明该位是 1。接着，我们按照上述步骤对每一位进行猜测，128 次迭代后即可恢复出完整的 AES 密钥，该过程在时间成本上是有效的。

3.2.2 代码分析

WUP、*client* 和 *server* 类的构造

基于 3.2.1 节中的实现原理，我们分别对 WUP、*client* 和 *server* 的类进行构造。需要注意的是，AES 密钥的生成由 *client* 完成，并将生成的密钥通过参数传给 *server* 类的构造函数。

```
1 # WUP class
2 class WUP:
3     def __init__(self, request, response, key, bytes_length):
4         self.request = request
5         self.response = response
6         self.encrypted_key = key
7         self.bytes_length = bytes_length
8         self.crc = "10110011"
9
10 # Client class
11 class client:
12     def __init__(self, key_size):
13         # Keys generated by textbook_RSA
```

```

14     self.public_key, self.private_key = read_keys()
15     # AES key
16     self.AES_key = random.randrange(1 << (key_size - 1), 2 ** key_size)
17     while self.AES_key % 2 == 0:
18         self.AES_key = random.randrange(1 << (key_size - 1), 2 ** key_size)
19     # Encrypt the plaintext
20     def encrypt_plaintext(self, plaintext, key_size):
21         ...
22     # Decrypt the ciphertext
23     def decrypt_ciphertext(self, ciphertext):
24         ...
25
26 # Server class
27 class server:
28     def __init__(self, AES_key):
29         # Keys generated by textbook_RSA
30         self.public_key, self.private_key = read_keys()
31         # AES key
32         self.AES_key = AES_key
33     # Encrypt the plaintext
34     def encrypt_plaintext(self, plaintext, key_size):
35         ...
36     # Decrypt the ciphertext
37     def decrypt_ciphertext(self, ciphertext):
38         ...

```

WUP 历史记录的生成

模拟 WUP 交互报文中的 `request` 和 `response` 字段分别来自 `"./test/request.txt"` 和 `"./test/response.txt"` 文件，需要对相应的 `txt` 文件进行读取。本模块的 `AES` 加密过程使用了 `Crypto.Cipher` 中实现的 `AES` 加密函数。

首先，利用 `client` 生成的 `AES` 会话密钥来实例化一个 `ECB` 模式的 `AES` 加密器：

```

1 # AES Encryptor with ECB mode
2 AES_encryptor = AES.new(a2b_hex(hex(AES_key)[2:]), AES.MODE_ECB)

```

接着，分别对 `response` 和 `request` 信息进行 `AES` 加密：

```

1 # Encrypt request with AES
2 message.request = bytes2bits(b2a_hex(AES_encryptor.encrypt(request.encode('utf-8'))))
[2:]
3 ...
4 # Encrypt response with AES
5 message.response = bytes2bits(b2a_hex(AES_encryptor.encrypt(response.encode('utf-8'))))
[2:]

```

最后，将 `AES` 会话密钥利用基于公钥的模指数计算进行 `RSA` 加密：

```
1 # Encrypt AES with public key in RSA
2 message.encrypted_key = fast_exp_mod(AES_key, public_key[1], public_key[0])
```

将上述三个字段组合成一个 *WUP* 交互报文对象，并返回。

AES 密钥的破解

根据 3.2.1 节中给出的 *CCA2* 实现原理，我们对 *AES* 密钥的破解进行函数实现。首先，设置一个 *for* 循环来对密钥的每一位进行迭代猜测。在每次迭代中，先将最高位设置为 1，再用其进行 *AES* 加密，并将加密后的 *request* 字段及 *AES* 密钥通过 `query_decryptor()` 函数访问解码机，获取解密后的明文结果。若能够正确解码，则说明该位的确是 1；否则，设置为 0。

```
1 # Break AES key
2 def break_AES_key(fake_request, public_key, private_key, message_encrypted_key,
key_size):
3     print("Breaking AES key...")
4
5     # AES key
6     AES_key = 0
7     # Traverse each bit
8     for i in range(key_size, 0, -1):
9         # 最高位设置为1
10        trial_key = int(AES_key >> 1) + (1 << (key_size - 1))
11        # AES Encryptor with ECB mode
12        AES_encryptor = AES.new(a2b_hex(hex(trial_key)[2:]), AES.MODE_ECB)
13        # Encrypted request
14        encrypted_request =
15        str(b2a_hex(AES_encryptor.encrypt(fake_request.encode('utf-8'))), encoding='utf-8')
16        # Encrypted key
17        factor = fast_exp_mod(2, (i - 1) * public_key[1], public_key[0])
18        encrypted_key = fast_exp_mod(message_encrypted_key * factor, 1, public_key[0])
19        # Plain request from querying decryptor (CCA)
20        plain_request = query_decryptor(encrypted_request, encrypted_key, private_key,
key_size)
21        try:
22            plain_request = plain_request.decode().strip(b'\x00'.decode())
23        except UnicodeDecodeError:
24            plain_request = "error"
25        if plain_request == "We attempt to perform CCA2 attack on textbook RSA":
26            AES_key = trial_key
27        else:
28            # 最高位设置为0
29            trial_key = int(AES_key >> 1)
30            AES_key = trial_key
31    return AES_key
```

WUP 交互报文的解密

在获取 AES 会话密钥后，攻击者调用 `decrypt_WUP_message()` 函数，首先实例化一个基于破获的 AES 密钥的 AES 解码器，再对 WUP 交互报文中的 *request* 及 *response* 字段进行解密，并打印结果。

```
1 # Decrypt WUP message using AES key
2 def decrypt_WUP_message(message, AES_key):
3     # Bits to bytes
4     bytes_request = bits2bytes(message.request, message.bytes_length)
5     bytes_response = bits2bytes(message.response, message.bytes_length)
6
7     AES_decryptor = AES.new(a2b_hex(hex(AES_key)[2:]), AES.MODE_ECB)
8     plain_request = str(AES_decryptor.decrypt(a2b_hex(bytes_request)), encoding='utf-8').rstrip("\0")
9     plain_response = str(AES_decryptor.decrypt(a2b_hex(bytes_response)), encoding='utf-8').rstrip("\0")
10    print("")
11    print("##### Decrypted History WUP Message Info")
12    print("- Decrypted request: ", plain_request)
13    print("- Decrypted response: ", plain_response)
14
15    print("#####")
```

3.2.3 模块使用方式

- 若需要更新 *client* 和 *server* 之间的 RSA 密钥，请参考 *task 1* 中重新生成指定 *size* 的密钥
- 指定 AES key 的大小，模拟在 *client* 和 *server* 之间的 CCA2 attack:

```
1 | python CCA2_attack.py --key_size 128
```

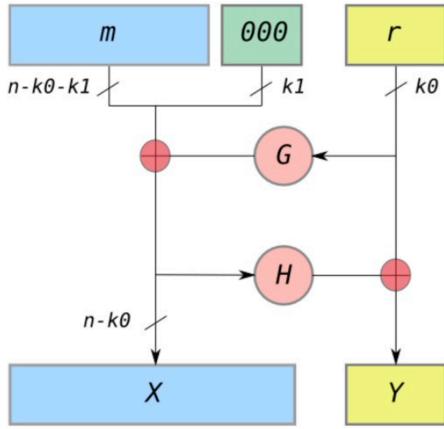
3.3 OAEP 密钥填充策略

3.3.1 实现原理

OAEP 密钥填充策略使用两个哈希函数 G 和 H ，输入为明文消息 m ，一固定个比特长度的随机数 r ，以及为验证消息而填充的 0 冗余串。其变换过程可通过下式来表示：

$$OAEP(m, r) = (X||Y) = (m||0^{k_1}) \oplus G(r)||r \oplus H((m||0^{k_1}) \oplus G(r))$$

其中， $X = (m||0^{k_1}) \oplus G(r)$ ， $Y = r \oplus H((m||0^{k_1}) \oplus G(r))$



当 OAEP 所基于的陷门单向置换 f 是单向安全时，OAEP 在随机预言模型下可证明达到适应性选择密文攻击下的不可区分安全性。同时，OAEP 的计算性能和空间利用率在实际应用中都非常有效。

3.3.2 代码分析

填充/反填充函数的实现

首先，我们基于论文 [3] 分别实现了 OAEP 的 *padding* 和 *unpadding* 函数，注意，这里默认 RSA 密钥长度为 1024-bit，且消息的比特长度要小于密钥长度（若大于，可以进行适当的划分）。我们利用 `hashlib` 工具包提供的 `sha512` 创建哈希函数，并使用 `random.SystemRandom()` 来随机初始化输入的随机数。此外，我们在实现中设置 $k_0 = 256$ ，即随机数的比特长度为 256-bit。

```

1 # Padding
2 def binary_padding(msg, key_size):
3     # Oracle
4     oracle_1 = hashlib.sha512()
5     oracle_2 = hashlib.sha512()
6     # Rand bits
7     rand_bits = format(random.SystemRandom().getrandbits(k0_bits_int), k0_bits_fill)
8
9     if len(msg) <= (key_size - k0_bits_int):
10         k1_bits = key_size - k0_bits_int - len(msg)
11         padded_msg = msg + ("0" * k1_bits)
12     else:
13         padded_msg = msg
14     # Update oracles
15     oracle_1.update(rand_bits.encode('utf-8'))
16     x = format(int(padded_msg, 2) ^ int(oracle_1.hexdigest(), 16), '0768b')
17     oracle_2.update(x.encode('utf-8'))
18     y = format(int(oracle_2.hexdigest(), 16) ^ int(rand_bits, 2), k0_bits_fill)
19     return x + y, len(msg)
20
21 # Unpadding
22 def binary_unpadding(padded_msg, n_bits):
23     # Oracle
24     oracle_1 = hashlib.sha512()
25     oracle_2 = hashlib.sha512()
26     x = padded_msg[0:768]
27     y = padded_msg[768:]
28     oracle_2.update(x.encode('utf-8'))

```

```

29     r = format(int(y, 2) ^ int(oracle_2.hexdigest(), 16), k0_bits_fill)
30     oracle_1.update(r.encode('utf-8'))
31     msg = format(int(x, 2) ^ int(oracle_1.hexdigest(), 16), '0768b')
32     msg = msg[0: n_bits]
33     return binary_to_dec(msg)

```

OAEP 填充/反填充函数的实现

在上述填充函数的基础上，我们实现了完整进行 OAEP 密钥填充和反填充的函数，用于对消息进行操作。

```

1 # OAEP key padding
2 def OAEP_key_padding(plaintext, key_size):
3     # Transfer to binary
4     bits = dec_to_binary(plaintext)
5     # Padding
6     padded_msg, n_bits = binary_padding(bits, key_size)
7     # New plaintext
8     plaintext = binary_to_dec(padded_msg)
9     return plaintext, n_bits
10
11 # OAEP key unpadding
12 def OAEP_key_unpadding(plaintext, n_bits):
13     # Transfer to binary
14     padded_msg = dec_to_binary(plaintext)
15     # # Padding
16     # New plaintext
17     plaintext = binary_unpadding(padded_msg, n_bits)
18     return plaintext

```

3.3.3 模块使用方式

- 基于 `textbook_RSA.py` 进行效果验证：

- 若进行 *padding* 但不进行 *unpadding*（模拟被攻击者截取，无法 *unpadding*）：

```

1 python textbook_RSA.py --encrypt_file ./test/plaintext.txt --OAEP_key_padding
2 python textbook_RSA.py --decrypt_file ./test/ciphertext.txt

```

- 同时进行 *padding* 和 *unpadding*（模拟正常的接收者）：

```

1 python textbook_RSA.py --encrypt_file ./test/plaintext.txt --OAEP_key_padding -
-decrypt_OAEP
2 python textbook_RSA.py --decrypt_file ./test/ciphertext.txt

```

- 在 *CCA2 attack* 中进行效果验证：

- 若进行 *padding* 但不进行 *unpadding*（模拟被攻击者截取，无法 *unpadding*）：

```

1 python CCA2_attack.py --OAEP_key_padding

```

- 同时进行 padding 和 unpadding (模拟正常的接收者) :

```
1 | python CCA2_attack.py --OAEP_key_padding --decrypt_OAEP_for_receiver
```

4. 实验结果

4.1 Textbook RSA

密钥生成

调用相关指令进行指定大小的 RSA 密钥的生成。可以看到，该操作成功生成了一对 $N = 1024$ 的 RSA 公钥和私钥，并保存到 `./keys/` 目录下。

```
Textbook_RSA_Experiments -- zsh - 140x40
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python textbook_RSA.py --generate_keys --key_size 1024
----- Begin Generating Keys -----
Generating big primes (may take some time) ...
Try 0 for the generation of big primes...
Calculating Euler phi function of N...
Formulating public key and private key...
Keys generation if completed! | Public key: (N, 11893276460897155733761191249390021249821609266969359141445097651844702946407560765247865730
4914263702452813080397630646651513083031406041056903262899851261889844234368664129453334279004887925407202785096181359414742388076837221540
012240637308083525894608828018014171132898910459070546193627943227686939763) | Private key: (N, 3647018670266704043655861174856214584412691
509985238642797618938596313356086528517832624807928115911841652809296141240275141468727041914650049191465010460579826601417948363132561
931207316720071411423015786182142561888460763433789095716395105395939868931897968938964477961793448589947390985719883848491)
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments %
```

文件加密

调用相关指令进行指定路径下 `txt` 文件的加密。可以看到，该操作成功将 `./test/plaintext.txt` 文件中的多行文本进行了 RSA 加密，并将结果写入 `./test/ciphertext.txt` 文件中。

```
Textbook_RSA_Experiments -- zsh - 140x40
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python textbook_RSA.py --encrypt_file ./test/plaintext.txt
----- Begin Encrypting plaintext -----
Encrypted result: [4515487334337858884087881783941858886720395698397268208460826611415703000142482050782289758460023723392895331074668470622
22337673916655085461166888381818973105376040719224213641992141043179987887645281175343083595982628794438910745003946454189730103232248919546
24483575809306053108333047496559810281854964334, 4859629876577601574933490138715655140220513862477950029546830500371350278381643560680722712
72395191295301807844861511951488484891498188002776752957163929581352069705005358801136578896110681557664943932892296927427204491207904973000
63243261479899239029279345474871062736406721750037702470082449406667420867966, 4903115388936561253403310171649175666990434645846416099533146
82595361375376022677364909334339940504742583803523120222763960436052095750360654977873781899468886507302784817099203153306941749214338033480
26608534370440235775694771798877055748477064294601664611824903879862926412659817822244939921129768056624563]
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments %
```

文件解密

完成加密后，调用相关指令进行指定路径下 `txt` 文件的解密。可以看到，该操作成功将 `./test/ciphertext.txt` 文件中的密文解密为明文，并写入 `./test/plaintext.txt` 中。

```
Textbook_RSA_Experiments -- zsh - 140x40
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python textbook_RSA.py --decrypt_file ./test/ciphertext.txt
----- Begin Decrypting plaintext -----
Decrypted result: ['123456', 'abcd', '@#$%^&*']
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments %
```

4.2 针对 *textbook RSA* 的 CCA2 攻击

调用相关指令进行 CCA2 攻击的模拟。注意，指令通过 `--key_size` 参数设置 AES 密钥的大小。可以看到，对于原始的 `request` 和 `response` 请求，`client` 使用生成的 AES 密钥来进行加密，生成历史 WUP 交互报文。我们模拟的攻击者在获得 WUP 历史交互报文后，通过运行 3.2.2 节中实现的 `break_AES_key()` 函数来进行 AES 密钥的攻破，并打印结果。获取 AES 密钥后，攻击者调用 `decrypt_WUP_message()` 函数来进行历史 WUP 交互报文中的字段解密，将解密结果输出并打印。

4.3 OAEP 密钥填充策略

基于 *textbook RSA.py* 的验证

我们首先模拟在添加 *OAEP padding* 后，攻击者截取 WUP 交互报文并窃取了 RSA 私钥后的结果。可以看到，虽然攻击者能够直接使用 RSA 私钥，但若不添加 *OAEP unpadding* 的操作，仍无法进行会话密钥的解密。

```
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python textbook_RSA.py --encrypt_file ./test/plaintext.txt --OAEP_key_padding
----- Begin Encrypting plaintext -----
Encrypted result: [852438952705274690308868843070484217976870966876138500889826315245756114909567285021491565618162732545713719786946499812960750899986368051589954832822025815998539446062873285444127559801285485942442848142442502229082186266869085385376413754036883365812973056201139789436717020327614267182601624624023349685, 88912339051643399090449649666580866837571733161222657194854324850027065928885007937289952886642114722995684441163737510787610032473130275763261628549204056784923177048632592755266939774356954223448960532186031524882406941419220925688790581228566413225090003463893792697552279310459712916998675591211563169, 24076383113968006290893063209435080938390088133917374515171855572411775201194668920372966500507435759290571891101635751296929186355278124859444975781689207510207889296824536068682626152918778172677966142474525908187968361485949898760126270934007922230259618930687764844345445263181299971794487957319694312733]
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python textbook_RSA.py --decrypt_file ./test/ciphertext.txt
----- Begin Decrypting plaintext -----
Traceback (most recent call last):
  File "textbook_RSA.py", line 51, in <module>
    textbook_RSA_func()
  File "textbook_RSA.py", line 47, in textbook_RSA_func
    decrypt_ciphertext(private_key, file_path, output_path)
  File "/Users/dicardo/PycharmProjects/NetworkLayerAttacks/Textbook_RSA_Experiments/utils.py", line 291, in decrypt_ciphertext
    plaintext = str(plaintext, encoding='utf-8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x97 in position 3: invalid start byte
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % |
```

接着，我们验证在添加了 *OAEP unpadding* 之后，真正的接收方能够对密文进行正常的解密。可以看到，在添加了 `--decrypt OAEP` 字段后，真正的接收方能够解出 AES 会话密钥并进行字段的解码。

```
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python textbook_RSA.py --encrypt_file ./test/plaintext.txt -oAEP_key_padding --decrypt_oAEP
----- Begin Encrypting plaintext -----
Encrypted result: [4515487334337858884087881783941858886720395698397268208460826611415703000142482050782289758460023723392895331074668470622
22337673916655085461166888381818973105376040719224213641992141043179987887645281175343083595982628794438910745003946454189730103232248919546
24483575809306053108333047496559810281854964334, 48596298765776015749334901387156551402205138624779500295468305003713502783816435606807272712
7239519129531080784486151195148848891498188002776752957613692958135206970500535880113657889611068155766494393289296927427204491207904973000
634232614798992390297293454748710627364067215003770247082449406674720867966, 49031153889365125340331017164917566699043464584616099533146
82595361375376022677364909334339940504742583803523120222763960436052095750360654977873781899468886507302784817099203153306941749214338033480
26608534370440235775694771798877055748477064294601664611824903879862926412659817822244939921129768056624563]
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python textbook_RSA.py --decrypt_file ./test/ciphertext.txt
----- Begin Decrypting plaintext -----
Decrypted result: ['123456', 'abcd', '@#$%^&*']
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % |
```

在 *CCA2 attack* 中进行效果验证

我们首先验证在仅进行 OSAP 密钥填充时，CCA2 的攻击结果。注意，此时模拟的是发起 CCA2 攻击的攻击者。可以看到，在攻击者试图进行 CCA2 攻击时，无法进行 WUP 历史交互报文的字段解码。

```
Textbook_RSA_Experiments -- zsh -- 140x40
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % python CCA2_attack.py --OAEP_key_padding
----- Begin Generating history WUP message -----

----- Performing CCA2 attack on textbook RSA -----

Error occurred when decrypting WUP message...exit
(NetworkLayerAttacks) dicardo@xuechunyudeMacBook-Pro Textbook_RSA_Experiments % |
```

接下来，我们验证在 CCA2 攻击模型下，合法的接收方仍能够通过 *OAEP unpadding* 获取正常的解码结果。当添加 `--decrypt_OAEP_for_receiver` 参数后，接收方能够在 CCA2 攻击模型下进行正常的会话密钥解码，并以此解密 WUP 交互报文。

5. 实验心得

在本次实验中，我首先在课堂授课内容的基础上，系统性地学习了 *textbook RSA* 的理论知识，并掌握了其在代码层面上的实现方法；其次，通过阅读论文 [2] 中给出的自适应选择密文攻击方案，我得以实现针对 *textbook RSA* 的 CCA2 攻击；最后，通过复现论文 [3] 中提到的 OAEP 密钥填充策略，我成功实现了 *textbook RSA* 的安全性改善，使其能够抵御 CCA2 级别的攻击。

在实验中，虽然在从算法到代码的复现的过程中遇到了一些困难，但经过不断的查阅资料和分析问题，我得以将这些问题一一解决，并从中收获良多。例如，在进行密文/明文的格式转换时，经常会遇到解码错误的报错，通过在代码中添加一些异常处理语句（如 *try* 语句）的方式才得以解决。

整个项目约花费一天半的时，在本次试验中，我不仅深入了解了 *textbook RSA* 的相关理论知识和复现方法，还掌握了如何实现对存在漏洞的加密方案进行安全攻击的方法，以及应对这种攻击的改进措施。在这个过程里，我不仅提高了自身发现 *bug*、解决 *bug* 的能力，更增强了自身从理论知识到实践结果的复现能力。

最后，感谢刘振、朱浩瑾老师在课堂及课后答疑上的教学和指导，感谢孙随彬助教在课程大作业上的解答和指导！

References

[1] RSA in Wikipedia: [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

[2] Knockel J, Ristenpart T, Crandall J. When textbook RSA is used to protect the privacy of hundreds of millions of users[J]. arXiv preprint arXiv:1802.03367, 2018.

[3] Rogaway P. Optimal asymmetric encryption how to encrypt with rsa[J]. 1995.

[4] [1]王巍, 高峻, 刘杰. OAEP 及其改进协议研究[J]. 信息安全与通信保密, 2009.