# IN4191 Security and Cryptography
# Assignment 3

Eric Camellini
4494164
e.camellini@student.tudelft.nl

9 October 2015

**Abstract**

In this report I explain the chinese remainder theorem (CRT) and how it can be used to optimize RSA decryption. I also show an example program that performs AES encryption and AES secret key exchange using RSA. To conclude, I analyze the RSA decryption time measured in the program, comparing the optimized (with the CRT) and the standard methods. I personally wrote the RSA implementation used in this example.

## 1 Introduction

### 1.1 Chinese remainder theorem statement

Given a sequence $n_1 \cdots n_k$ of positive integers, if they are pairwise coprime then, for any given sequence of integers $a_1 \cdots a_k$, there exists an integer x solving the following system of simultaneous congruences:

$$\begin{cases} x = a_1 \pmod{n_1} \\ \quad \vdots \\ x = a_k \pmod{n_k} \end{cases}$$

Furthermore:

$$x = y \pmod{n_i}, \quad 1 \leq i \leq k \qquad \Longleftrightarrow \qquad x = y \pmod{N}, \quad N = n_1 \cdots n_k.$$

[5][3]

### 1.2 CRT example

In this section I provide a numerical example of the CRT. The problem to solve is the following:

$$\begin{cases} x = 2 \pmod{3} \\ x = 2 \pmod{4} \\ x = 1 \pmod{5} \end{cases}$$

$3, 4, 5$ are pairwise coprime because $gcd(3, 4) = 1, \quad gcd(3, 5) = 1, \quad gcd(4, 5) = 1$, so we can apply the CRT. The theorem says that a solution exists and that it is in $\pmod{3 \cdot 4 \cdot 5} = \pmod{60}$. In this case the solution is $26 \pmod{60}$ (it means that there are many solutions to the system, of the form $26 + k \cdot 60, \quad k = 0, 1, 2 \cdots$)

### 1.3 CRT for RSA optimization

The idea behind the optimization is to split the RSA decryption modular exponentiation into two smaller ones, each one used for one half of the message[4]. This can be done only knowing the factorization $n = pq$, so only who has the private key can use this approach. The procedure is the following (see also [6]):

1. Compute:
$$d_P = d \pmod{p-1}$$
$$d_Q = d \pmod{q-1}$$
$$q_{\text{inv}} = q^{-1} \pmod{p}$$
2. Compute a CRT representation of the received ciphertext $c$ (this can be done because $p$ and $q$ are coprime):
$$m_1 = c^{d_P} \pmod{p}$$
$$m_2 = c^{d_Q} \pmod{q}$$
3. Use the following formula to build the message:
$$h = q_{\text{inv}}(m_1 - m_2) \pmod{p}$$
$$m = m_2 + h \cdot q$$

## 2 Methods

To test the optimization described in Section 1.3 in I built an example program that simulates a simple secret sharing scenario using AES (symmetric cipher) with key distribution through RSA (asymmetric cipher):

1. Bob generates an RSA key pair;
2. Alice sends and AES-128 encrypted message to Bob;
3. Alice, in order to allow Bob to decrypt the messages that she sends, encrypts the AES secret key with RSA using Bob's public key, and sends it to him.
4. Bob receives the encrypted secret key, he decrypts it with its private key and he can then use it to decrypt the message sent by Alice.

I personally wrote the RSA implementation used in this program and I used a library ([2]) for prime and random numbers generation, modular arithmetic and for the AES implementation (see the appendices for details).

The program performs the RSA decryption both with and without the CRT optimization, and measures all the encryption and decryption execution times. In order to analyze the performance of the two decryption procedures I made some tests with different bit sizes for $n$ and different payloads. I also made a program that runs the described example for a selected amount of times and calculates the average of the various execution times.

## 3 Results

An example of program execution with a 1024-bits $n$ generated in the RSA algorithm is shown in Figure 1, it includes also the execution times. The average execution times over 1000 iterations, again with a 1024-bits $n$, are shown in Figure 2.

See the appendices for the source code and for some implementation details.

```
___BOB___
BOB generates an RSA key pair:
TIMING: RSA key pair generation: 36.988974 ms
Bob's public key:
(139909029570260277493920294367516720587093318497072175111454493980956276697647056477522
162866797960797908694647765830712472264929736353282690697314442905103835291894603029126 5

___ALICE___
Alice's message (input):  Alice's input message
TIMING: AES-128 encryption: 0.093937 ms
AES-128 ciphertext:  B4yLqXbi7oeJx9vSUVJ3wEzzrQydFXqe+jDl8gflsrKKrXg37X5kcJ9pBJ/G0hid
TIMING: RSA encryption: 72.728157 ms
Encrypted AES key:
MDI3MDY5MjA4MjYxNjMxNTU2MjU0NTY5MTY4NzYyNDI3MDU0MjM2Mjc0NTI3MzAzNDMyNzA4MjE1ODc3MzcyODcz

___BOB___
TIMING: RSA decription (no crt): 85.749149 ms
TIMING: RSA decryption (crt): 83.138943 ms
TIMING: AES decryption: 0.037909 ms
Received plaintext:  Alice's input message
```

Figure 1: Example of program execution. The RSA public key and the AES encrypted key have been cut to make the image more readable, execute the code in the appendices to see a complete output.

```
AVG times over 1000 executions:
KEY generation time: 30.538122 ms
AES encryption time: 0.082491 ms
RSA encryption time: 73.450027 ms
RSA decryption time: 89.382718 ms
RSA + CRT dec. time: 85.549324 ms
AES decryption time: 0.035302 ms
```

Figure 2: Average execution times over 1000 executions

# 4 Discussion

Accordingly to the time measurements that I performed, the CRT decryption method always behaves more efficiently: on average it reduces the execution time of the 6% (see Figure 2).

From a general analysis of this Python's RSA implementation we can see that the operations that involve modular exponentiation are efficient (the Python built-in *pow* function points to underlying C libraries as stated in [1]). In a first version of the program I tried to implement RSA from scratch, including the generation of the prime numbers and the modular arithmetic functions, but using the *pow* Python function for exponentiation (like I do in the described final version). The bottleneck of this first attempt was the RSA key pair generation: even with an efficient implementation of the Sieve of Erastothenes, used to generate the two prime numbers, the operation used to take seconds with a bit size (of $n$, the modulo) greater than 18, more than 1 minute with a bit size of 26, and my computer was not able to handle greater bit sizes. In the end I decided to use a library also for the numbers generation part and for some modular operations (e.g. inverse).

# 5 Conclusion

In this work I successfully implemented an efficient RSA module in python: it also includes the support for an optimized decryption that uses the CRT. I built an example program that perform AES encryption and

AES secret key exchange using RSA, comparing the execution times of two RSA decryption techniques: the standard and the CRT one. The results show that the CRT version is, on average, 6% more time efficient than the classic method.

## References


[1] Modular exponentiation in python. http://aditya.vaidya.info/blog/2014/06/27/modular-exponentiation-python/. Accessed: October 19, 2015.

[2] Pycrypto - the python cryptography toolkit. https://www.dlitz.net/software/pycrypto/. Accessed: October 19, 2015.

[3] Randell heyman - the chinese remainder theorem made easy. https://www.youtube.com/watch?v=ru7mWZJlRQg. Accessed: October 19, 2015.

[4] Using the crt with rsa. http://www.di-mgt.com.au/crt_rsa.html. Accessed: October 19, 2015.

[5] Wikipedia - chinese remainder algorithm. https://en.wikipedia.org/wiki/Chinese_remainder_theorem, note = Accessed: October 19, 2015.

[6] Wikipedia - rsa - using the chinese remainder algorithm. https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Using_the_Chinese_remainder_algorithm. Accessed: October 19, 2015.


# Appendices

The appendices contain the source code (Python 2.7) of all the modules that I built in this work. The link to references used are inserted in the code as comments.

## A aes.py

In this module I implemented the AES chiper building an AESCipher class. It is based on the PyCrypto library and it used the CBC (cipher block chaining) AES mode. This implementation is based on the two that I linked in the source code. The encryption outputs a base64 encoding of the ciphertext and the decryption requires an input text with the same encoding to work.

```
import base64
from Crypto.Cipher import AES
from Crypto import Random


#stackoverflow.com/questions/12524994/encrypt-decrypt-using-pycrypto-aes-256
#stackoverflow.com/questions/12562021/aes-decryption-padding-with-pkcs5-python
BS = AES.block_size
_pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
_unpad = lambda s: s[:-ord(s[len(s)-1:])]


class AESCipher:
    def __init__(self, key):
        self.key = key

    def encrypt(self, raw):
```

```
        raw = _pad(raw)
        iv = Random.new().read(BS)
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return base64.b64encode(iv + cipher.encrypt(raw))

    def decrypt(self, enc):
        enc = base64.b64decode(enc)
        iv = enc[:16]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return _unpad(cipher.decrypt(enc[16:]))
```

# B rsa.py

In this module I implemented the RSA chiper building an RSACipher class. An instance of the class can be used in for:

1. Encryption with the public key (e, n): in this case the public key used for the encryption must be set using the *set_public_key* method, then the *encrypt* method can be used.
2. Key pair generation and decryption using the private key: in this case a key pair must be generated using the *gen_key_pair* method, then the *decrypt* method can be used. The corresponding public key can be obtained (to be distributed) using the *get_public_key* method.

The module provides also methods for string encryption and decryption: these methods act on the input string character by character and the result of the encryption is a base64 encoded string (same format expected for the decryption). It is not the more efficient way to encrypt/decrypt a string, but for the purpose of comparing the efficiency of two decryption methods it is acceptable.

```
import base64
from Crypto.Util import number


class RSACipher:

    def __init__(self):
        self.e = None
        self.d = None
        self.n = None
        None

    def set_public_key(self, (e, n)):
        self.e = e
        self.n = n
        self.block_size = len(str(n))

    def _gen_primes(self, bit_length):
        self.p = number.getStrongPrime(bit_length)
        self.q = number.getStrongPrime(bit_length)

    def gen_key_pair(self, bit_length):
        if(bit_length % 2 != 0):
            raise ValueError("Bit length must be a multiple of 2")

        if(bit_length < 8):
            raise ValueError("Bit length must be at least 8 bits")
```

```
        bits_pq = bit_length/2

        self.n = 0
        while (self.n.bit_length() != bit_length):
            self._gen_primes(bits_pq)
            self.n = self.p*self.q

        self.phi_n = (self.p-1)*(self.q-1)

        self.e = 0
        while((self.e <= 3) |
              (self.e >= self.phi_n) |
              (number.GCD(self.e, self.phi_n) != 1)):
            self.e = number.getRandomInteger(self.phi_n.bit_length())

        self.d = number.inverse(self.e, self.phi_n)
        self.block_size = len(str(self.n))

        #Initializing the CRT values
        self._init_crt()

    def _init_crt(self):
        self.dp = self.d % (self.p - 1)
        self.dq = self.d % (self.q - 1)
        self.q_inv = number.inverse(self.q, self.p)

    def get_public_key(self):
        if (self.e is None) | (self.n is None):
            raise UnboundLocalError("Key pair not generated.")
        return (self.e, self.n)

    def encrypt(self, m):
        if (self.e is None) | (self.n is None):
            raise UnboundLocalError("Encryption public key not set.")
        return pow(m, self.e, self.n)

#crypto.stackexchange.com/questions/2575/chinese-remainder-theorem-and-rsa
#en.wikipedia.org/wiki/RSA_(cryptosystem)#Using_the_Chinese_remainder_algorithm
    def decrypt(self, c, crt=False):
        if (self.d is None) | (self.n is None):
            raise UnboundLocalError("Key pair not generated.")
        if crt is True:
            m1 = pow(c, self.dp, self.p)
            m2 = pow(c, self.dq, self.q)
            h = (self.q_inv * (m1 - m2)) % self.p
            return m2 + h*self.q
        else:
            return pow(c, self.d, self.n)

    def encrypt_string(self, s):
        ciphertext = ""
        for c in s:
            e_c = str(self.encrypt(ord(c)))
            while len(e_c) < self.block_size:
                e_c = '0' + e_c
            ciphertext += e_c
        return base64.b64encode(ciphertext)

    def decrypt_string(self, s, crt=False):
```

```
        s = base64.b64decode(s)
        plaintext = ""
        block = ""
        for c in s:
            if(len(block) < self.block_size):
                block += c
            else:
                plaintext += chr(self.decrypt(int(block)))
                block = c
        plaintext += chr(self.decrypt(int(block), crt))
        return plaintext
```

# C example.py

This is the example program described in Section 2. To measure the execution times in an easy way I used the Python contextmanager module.

```
#!/usr/bin/python2.7
import sys
import aes
import rsa
import time
from contextlib import contextmanager


@contextmanager
def measure_time(label):
    t1 = time.time()
    yield
    t2 = time.time()
    print 'TIMING: %s: %0.6f ms' % (label, (t2-t1)*1000)


if __name__ == "__main__":
    if(len(sys.argv) != 2):
        print "Input missing. Program terminated."
        print "Usage: ./example.py 'input message'"
        sys.exit(0)
    else:

        ### BOB ###
        print "___BOB___"
        # Bob generates its RSA key pair"
        print "BOB generates an RSA key pair:"
        rsa_bob = rsa.RSACipher()
        with measure_time('RSA key pair generation'):
            rsa_bob.gen_key_pair(16)
        public_key_bob = rsa_bob.get_public_key()
        print "Bob's public key: ", public_key_bob
        print

        ### ALICE ###
        print "___ALICE___"
        # ALICE has an AES secret key, she encrypts it with Bob's public
        # key and she sends it to Bob. She sends also and AES encrypted
        # message
```

```
### AES-128 initialization, Alice has the key ###
key = "HardcodedKey?lol"  # 128-bits key
aes128 = aes.AESCipher(key)

input = sys.argv[1]  # reading and printing the message
print "Alice's message (input): ", input

### Step 1 - AES-128 encryption ###
with measure_time('AES-128 encryption'):
    ciphertext = aes128.encrypt(input)
print "AES-128 ciphertext: ", ciphertext

### Step 2 - RSA Key encryption
rsa_alice = rsa.RSACipher()
rsa_alice.set_public_key(public_key_bob)
with measure_time('RSA encryption'):
    e_key = rsa_alice.encrypt_string(key)
print "Encrypted AES key: ", e_key
print

### BOB ###
print "___BOB___"
# BOB receives the encrypted AES key, he decrypts it using
# its private RSA key and then he uses it to decrypt the
# secret message

### Step 3.1 - Bob receives and decrypt the secret key ###
with measure_time('RSA decription (no crt)'):
    key = rsa_bob.decrypt_string(e_key)
#print "Key: ", key

### step 3.2 - The same as 3.1, but with CRT optimization ###
with measure_time('RSA decryption (crt)'):
    key = rsa_bob.decrypt_string(e_key, True)
#print "Key (CRT): ", key

### Step 4 - AES-128 decryption ###
with measure_time('AES decryption'):
    plaintext = aes128.decrypt(ciphertext)
print "Received plaintext: ", plaintext
```

# D time_avg.py

This is the program that computes the execution time average over a fixed number of executions (1000 in this case), as described in Section 2.

```
#!/usr/bin/python2.7
import sys
import aes
import rsa
import time
from contextlib import contextmanager

ITERATIONS = 1000


@contextmanager
```

```
def measure_time(times_list):
    t1 = time.time()
    yield
    t2 = time.time()
    times_list.append((t2-t1)*1000)


def list_avg(l):
    return sum(l)/len(l)

if __name__ == "__main__":

    if(len(sys.argv) != 2):
        print "Input missing. Program terminated."
        print "Usage: ./time\_avg.py 'input message'"
        sys.exit(0)
    else:
        key_times = []
        aes_enc_times = []
        rsa_enc_times = []
        rsa_crt_dec_times = []
        rsa_dec_times = []
        aes_dec_times = []

        for i in range(ITERATIONS):
            print "Execution %d..." % i
            rsa_bob = rsa.RSACipher()
            with measure_time(key_times):
                rsa_bob.gen_key_pair(16)
            public_key_bob = rsa_bob.get_public_key()
            key = "HardcodedKey?lol"  # 128-bits key
            aes128 = aes.AESCipher(key)

            input = sys.argv[1]  # reading and printing the message
            with measure_time(aes_enc_times):
                ciphertext = aes128.encrypt(input)

            rsa_alice = rsa.RSACipher()
            rsa_alice.set_public_key(public_key_bob)
            with measure_time(rsa_enc_times):
                e_key = rsa_alice.encrypt_string(key)

            with measure_time(rsa_dec_times):
                key = rsa_bob.decrypt_string(e_key)

            with measure_time(rsa_crt_dec_times):
                key = rsa_bob.decrypt_string(e_key, True)

            with measure_time(aes_dec_times):
                plaintext = aes128.decrypt(ciphertext)

        print "Final plaintext: ", plaintext
        print "AVG times over %d executions:" % ITERATIONS
        print "KEY generation time: %0.6f ms" % list_avg(key_times)
        print "AES encryption time: %0.6f ms" % list_avg(aes_enc_times)
        print "RSA encryption time: %0.6f ms" % list_avg(rsa_enc_times)
        print "RSA decryption time: %0.6f ms" % list_avg(rsa_dec_times)
        print "RSA + CRT dec. time: %0.6f ms" % list_avg(rsa_crt_dec_times)
        print "AES decryption time: %0.6f ms" % list_avg(aes_dec_times)
```