CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Timing Attack on the RSA Cipher
Student:	Martin Andrýsek
Supervisor:	Ing. Jiří Buček
Study Programme:	Informatics
Study Branch:	Information Technology
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2018/19

Instructions

Review known timing side channel attacks on RSA decryption and signing operations. Create a demonstration application that will perform timing attack on RSA in order to determine the private key. The application will be used in courses on cryptology and computer security as a part of laboratory exercises. Consider an attack on a local computer or over the network and evaluate its time complexity.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 7, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Bachelor's thesis

Timing Attack on the RSA Cipher

Martin Andrýsek

Supervisor: Ing. Jiří Buček

15th May 2018

Acknowledgements

I would like to thank to Ing. Jiří Buček for leading me in this thesis and my friends for support me during writing.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 15th May 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Andrýsek, Martin. *Timing Attack on the RSA Cipher*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018. Also available from: ⟨`https://github.com/vydrous/BP-code`⟩.

Abstrakt

Tato práce se zabývá replikací útoku na RSA kryptosystém časovým postranním kanálem, který je realizován měřením času algoritmu opakovaných čtvercu s Montgomeryho násobením. Útok se zameřuje na měření času trvaní dešifrování rozdílných zpráv s určitými vlastnostmi. Práce popisuje základní principy a slabiny RSA kryptosystému. Výsledkem práce je demonstrativní aplikace, která bude pouzita ve vyuce předmetech, zabyvajícími se počítačovou bezpečností.

Klíčová slova RSA, kryptoanalýza, časový útok, postranní kanál, Montgomeryho násobení

Abstract

This thesis is focused on replication of timing attack on RSA cryptosystem introduced by Paul Kocher, which is done by measuring time of square and multiply algorithm with Montgomery multiplication. The attack is based on measuring execution time of decryption function on messages with different properties. The thesis describe main principles and vulnerabilities of RSA cryptosystem. Implementation should be used for education purposes, mainly in security courses.

Keywords RSA, cryptanalysis, timing attack, side channel, Montgomery multiplication

Contents

List of Figures

Introduction

Information security is nowadays very important. Through the network flows lot of information which is essential to keep in private. Due to this, many ciphers were invented and are used to encrypt communication over network. One of these ciphers is RSA. In past time there were several vulnerabilities on this cipher. I will focus on timing attack which exploit data dependency of decrypting algorithm.

This thesis will explain the main thoughts of RSA cryptosystem, its known vulnerabilities and how to defend against them. Thesis also introduce reader to timing attack problematic. It will compare two targets of timing attack, Kochers original attack on multiplication versus Dhems attack on square. Although both attack can be easily defended just by eliminating data dependency in decryption (resp. signing) algorithm.

Purpose of this thesis is demonstrative. The final application should be used in cryptography courses on Faculty of Information chnologies on CTU, mainly in Advanced Cryptology course. It should demonstrate progress of guessing private key bit by bit. There will be two approaches of guessing key. First original introduced by Paul Kocher which focuses on extra modural reduction during multiply operation[2]. The second approach was introduced by J.-F. Dhem and collective where they focus on extra modular reduction during square phase[3].

Lastly, I will introduce several ways of defense against timing attacks which will avoid data dependency. So that execution of time will be either constant or not telling us any usable information.

Chapter 1

State-of-the-art

In 1996 Paul Kocher presented timing attack on several cryptosystems including RSA[2]. The cryptosystems have in common that all of them are using modular exponentiation or they are public key cryptosystems. Kochers idea was to attack square and multiply algorithm which uses Montgomery multiplication. He intend to exploit execution time of decrypting and signing algorithms because there is dependency on private exponent. After Kocher there have been more tries with better success, for example J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater and J.-L. Willems who improved Kochers study[3].

Chapter 2

Used mathematical principles

In this chapter I will introduce mathematical theorems and principles which will be referenced through the thesis

2.1 Fermat's little theorem

Theorem 1 *(Fermat) "For prime p and any* $a \in Z$ *such that* $a \not\equiv 0 \pmod{p}$,

$$a^{p-1} \equiv 1 \pmod{p}"[4]$$

Fermat's little theorem is restricted only on prime numbers so that we will need to extend it to all integers. For that we will use Euler's theorem.

2.2 Euler's totient function

First we need to define Euler's totient function which is used in Euler's theorem.

Definition 1 *(totient) "For* $n \geq 1, \phi(n)$ *can be characterised as the number of postive integers less than n and relatively prime to it. The function* ϕ *is usually called the Euler totient function after its originator , (sometimes the phi-function)."[5]*

2.3 Euler's theorem

Theorem 2 *(Euler) "For* $m \geq 2 \in Z^+$ *and any* $a \in Z$ *such that* $\gcd(a, m) = 1$

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

where $\phi(m)$ *is the number of invertible integers modulo m."[4]*

2.4 Chinese remainder theorem

Theorem 3 *(CRT) "Let $m_1, ..., m_r$ be pairwise coprime natural numbers, and $a_i, (1 \leq i \leq r)$ be arbitrary integers. Write $M_i = \Pi_{j \neq i} m_j$.Let n_i be the multiplicative inverse of M_i modulo m_i. Then, the unique solution $N \bmod m_1 m_2 ... m_r$ to the system of congruences $N \equiv a_i \bmod m_i$ for all $i \leq r$ is given by*

$$N = a_1 n_1 M_1 + a_2 n_2 M_2 + ... + a_r n_r M_r."[6]$$

Chapter 3

RSA

RSA is public-key cryptosystem which was developed at MIT by Ron Rivest, Adi Shamir and Leonard Adleman. The cryptosystem was published in the 1977. [7]

Main thought of public-key cryptosystem is sharing encryption key to whoever asks for it. Beacause it is used just for encryption, we can be sure that attacker could not retrieve any information about secret decryption key. Each subject of the conversation keeps his decryption key in private.

3.1 Principle

RSA is based on modular exponentiation. There is three important numbers which are used in computing modular exponation resp. crypting messages.

- n - Modulus which is used during whole cipher algorith.
- e - Encryption exponent. Exponent e is used to modular exponation of message so that we get ciphertext. Exponent must meet the condition $\gcd(e, \phi(n)) = 1$ where ϕ is Euler's totient function and *gcd* is the greatest common divider function. This condition will be explained later.
- d - Decryption exponent. It is computed by finding modular inversion of public exponet e in modulus $\phi(n)$.

Commonly used value of e is 65537.

The cryprosystem consist of two keys, public and privat. Public key is used for encrypting messages and is composed of modulus n and public exponet e. Private key is composed of modulus n and private exponent d.

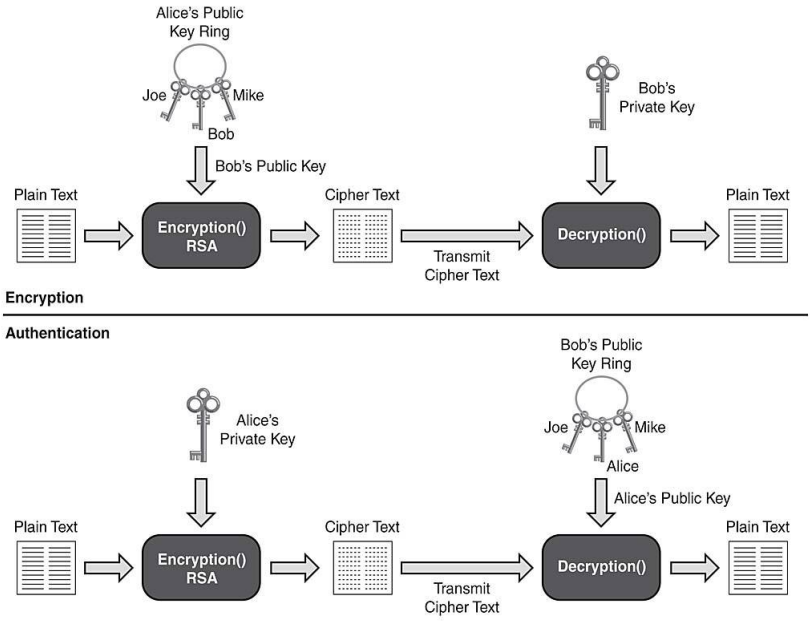

Figure 3.1: RSA illustration[1]

3.1.1 Key generation

This is steps needed to generate keypair including public and private key

- Generate p and q, which have to be distinct prime numbers large enough to make factorization of their product hard problem. If we choose too small p and q, we are risking that attacker could do factorization of n so that he could easily compute our private exponent d using Euler's totient function ϕ.
- Compute n, where $n = pq$
- Compute Euler's totient function $\phi(n)$. Because we know p and q it is simple to compute it. We know that n is product of p and q which are prime numbers. Euler's totient function ϕ for prime numers equals $\phi(P) = P - 1$ where P is prime number. Also we know that Euler's totient function ϕ is multiplicative. So the problem is:

  $$\phi(n) = \phi(p * q)$$

  Now we use multiplicative rule:

  $$\phi(p * q) \ = \phi(p) * \phi(q) = (p-1)(q-1)$$

  Then we get simple formula for comupting $\phi(n)$

  $$\phi(n) = (p-1)(q-1)$$

- Generate e from 3 to $n-1$ which meets condition $\gcd(e, \phi(n)) = 1$. In most implementation of RSA is used $e = 65537 = 2^{16} + 1$
- Compute $d = e^{-1} \bmod \phi(n)$
- The pair (e, n) is released as public key
- The pair (d, n) is kept secret as private key

3.1.2 Key distribution

- Alice would like to send Bob secret message.
- Bob generates public key (e, n) and his private key (d, n).
- Bob sends Alice public key using reliable route (it has not to be secret route).
- Due to high value of n possible attacker will not be able compute d from public keypair (e, n) because factorization of n is not possible in polynomial time.
- Alice uses it to encrypt her message and sends it to Bob.
- Bob decrypts her message using his private key.

3.1.3 Encryption

Encryption is done by using public keypair (e, n):

$$c = |m^e|_n$$

where m is plaintext message and c is encrypted message which will be sent to receiver.

3.1.4 Decryption

Decryption is done similar thanks to relation $ed \equiv 1 \pmod{\phi(n)}$. We can simply power ciphertext to our private exponent d to obtain original message.

$$|c^d|_n = |(m^e)^d|_n = |m^{ed}|_n$$

Now that we know, that $ed \equiv 1 \pmod{\phi(n)}$ which can be overwriten to form $ed = 1 + k * \phi(n)$. From that we get:

$$|m^{ed}|_n = |m^{1+k*\phi(n)}|_n = |m^1 * m^{k*\phi(n)}|_n$$

From Euler's theorem ($a^{\phi(n)} \equiv 1 \pmod{n}$) we can substitute $m^{k*\phi(n)}$ for 1.

$$|m^1 * m^{k*\phi(n)}|_n = |m^1 * 1|_n = m$$

We got desired message m.

3.1.5 Signing

RSA signing is used to verify wheter the message was not changed on the way from sender to reciever. The process is really similar to decryption but we do not encrypt message itself. We first use some one way function on message, mostly there is used some hash function, then we encrypt the hash of the message but we use our private exponent for encryption. That would mean anyone could decrypt the signature by using our public key and compare it with actual has of the recieved message. If they equals we can be sure that it is original message and was not changed on the way to from sender. If someone tries to change the content of message, he has to have sender private key to generate valid signature, otherwise he will not be able to generate signature from different hash (which has to change, when he change the content of message).

All content in this chapter to this point is based on [7].

3.2 Security

Security of RSA is relied on inability to do factorization of large integer in polynomial time. From paper "Factoring estimates for a 1024-bit RSA modulus"[8] we can say that breaking 1024-bit RSA key would last about year long on device worth of 10 milions american dolars.

There is a community challenge where people trying to factorize RSA keys of different length. The latest succces was on 768-bit length key which was factored in 2009. In that time the facotrization took 3 years of execution time. Nowadays standart key length is 1024 which have not been factored yet.[9]

3.3 Optimization

Because we generally use high value of modulus n the exponentiation of message of similar bit length is quite time consuming so several algorithms to increase speed of computation was developed.

3.3.1 Chinese remainder theorem

By using CRT we can significantly speed-up decryption of received messages or signing outcoming message. This method is not usable during encrypting phase because we need to know p and q factors of n which are parts of private key. Assuming that $p > q$ we can divide :

$$dP = e^{-1} \pmod{p-1}$$

$$dQ = e^{-1} \pmod{q-1}$$
$$qInv = q^{-1} \pmod{p}$$

After that, we compute message m with given c:

$$m_1 = c^{dP} \pmod{p}$$
$$m_2 = c^{dQ} \pmod{q}$$
$$h = qInv \cdot (m_1 - m_2) \pmod{p}$$
$$m = m_2 + hq$$

Finding modular exponentiation cost grows with cube of number of the bits in n, so it is still more efficient to do two exponentiation with half sized modulus

This whole section is based on [10].

3.3.2 Montgomery Multiplication

Normal modular multiplication could be quite slow for large numbers, due to processor have to run several operations before it gets desired remainder. On the other hand P. L. Montgomery developed algorithm which assumes that processor do division by power of 2 really fast.

Montgomery presented algorithm, which transform numbers to Montgomery base and then compute modular multiplication efficiently. To transform number to Montgomery base we need to compute $\bar{a} = ar \pmod{n}$ where r is the next power of 2 greater than n. For example if $2^{63} < n < 2^{64}$ then desired r will be 2^{64}. The multiplication in Montgomery base is done by:

$$\bar{u} = \bar{a}\bar{b}r^{-1} \pmod{n}$$

where r^{-1} is modular inversion of r.

As we can see $\bar{u}$ is in Montgomery base of the corresponding $u = ab \pmod{n}$ since

$$\begin{aligned} \bar{u} &= \bar{a}\bar{b}r^{-1} \pmod{n} \\ &= (ar)(br)r^{-1} \pmod{n} \\ &= (ab)r \pmod{n} \end{aligned} \tag{3.1}$$

Montgomery reduction which gives us $\bar{u}$ is implemented this way:

Algorithm 1 Montgomery Reduction

1: **function** MON_RED($\bar{a}, \bar{b}, N$)
2: $t \leftarrow \bar{a} * \bar{b}$
3: $m \leftarrow N^{-1} * t \pmod{r}$
4: $\bar{u} \leftarrow (t + mN)/r$
5: **if** $\bar{u} > N$ **then**
6: $\bar{u} \leftarrow \bar{u} - N$
7: **end if**
8: **return** $\bar{u}$
9: **end function**

Its main advance is that it never performs division by the modulus n but we still need to find out u and precompute n^{-1} using the extended Euclidean algorithm. It is done by this algorithm:

Algorithm 2 Montgomery Multiplication

1: **function** MON_MULT(a, b, n)
2: $r \leftarrow 2^{BitLen(n)}$
3: *Compute* n^{-1} *using the extended Euclidean algorithm*
4: $\bar{a} \leftarrow a * r \pmod{n}$
5: $\bar{b} \leftarrow b * r \pmod{n}$
6: $\bar{u} \leftarrow Mon\_Red(\bar{a}, \bar{b})$
7: $u \leftarrow Mon\_Red(\bar{u}, 1)$
8: **return** u
9: **end function**

This section derives from [11]

3.3.3 Square and Multiply

This optimization uses bitwise representation of the exponent. The algorithm picks all byte from left (MSB) to right and despite their value, it determines which operation will be performed for each bit. For bits equal to 1 we perform squaring preset value c then we multiply it with the base of exponentiation m. For bits equal to 0 we just perform squaring part. Therefore we get data dependent operation, which will be used in our attack. For even faster implementation we use Montgomery multiplication instead of normal one. In some papers this Square and Multiply algorithm is called Montgomery exponentiation

Algorithm 3 Square & Multiply algorithm

```
1: function SQUARE_AND_MULTIPLY(m, e, n)
2:     c ← 1
3:     k ← BitLen(e)
4:     for i ← k − 1, 0 do
5:         c ← Mon_Mult(c, c)
6:         if e[i] == 1 then                ▷ ith bit of exponent e
7:             c ← Mon_Mult(c, m)
8:         end if
9:     end for
10:    return c
11: end function
```

This section derives from [11]

Chapter 4

Attacks

The basic idea of timing attacks was presented by Kocher in 1996. He specified theoretical attacks not only on RSA. [2]

Both variant of attack are based on similar principle. They divide messages from set M to several subsets M_i due to response of some Oracle O. Then by measuring time of decrypting or signing and guessing bits of secret exponent by comparing times of each set.[2][12]

4.1 Attack on multiply

First Kochers idea was to exploit multiply operation in Square and Multiply algorithm. Kocher mean to measure time of decryption (or signing) messages using the private key d and focus on conditional multiply step. We are attacking each bit of d with knowledge of $i-1$ bits we can guess the ith bit. Let $d = d_1, d_2, \ldots, d_k$ where k is bit length of d and d_1 is MSB. We can assume that $d_1 = 1$ so we can attack bit d_2.[2]

We need oracle O which predict whether final Montgomery reduction happened during multiply step: [12]

$$O(m) = \begin{cases} 1 & \text{if } m^2 * m \text{ is done with final reduction} \\ 0 & \text{if } m^2 * m \text{ is done without final reduction} \end{cases}$$

where m is message from set M. We can now divide messages to 2 subsets:

$$M_1 = \{m \in M : O(m) = 1\}$$

$$M_2 = \{m \in M : O(m) = 0\}$$

We can now measure time of these two subsets. We are expecting same times for doing square part, but in multiply part will be messages from M_1 higher, due to final Montgomery Reduction. We compare means of sets M_1 and M_2. If time of M_1 is significantly bigger then the final reduction was done therefore bit d_2 is 1. If the times of M_1 and M_2 are equal then bit d_2 is 0. .

Problem: We cannot be sure what is significant difference between time means. So our guesses cannot be precise.[12]

4.2 Attack on square

Focusing on squaring operation will give us better results. The procedure is similar but we generate two oracles and four sets of messages. We similarly iterate through the bits of secret key d as in multiply attack. When we know $i-1$ bits and we are guessing ith bit we compute m_{temp} which has value before unknown possible multiplication step.[3]

We first presume that bit d_i is 1. If the presumption is right then the following steps will be executed. m_{temp} will be multiplied by m, then the result of multiplication will be squared. We will execute the multiplication step and then we will check if in the square step is done with or without reduction. By this criterion we divide messages to subsets M_1 if the reduction was computed or M_2 if not. The oracle will be: [3][12]

$$O_1(m) = \begin{cases} 1 & \text{if } (m_{temp} * m)^2 \text{ is done with final reduction} \\ 0 & \text{if } (m_{temp} * m)^2 \text{ is done without final reduction} \end{cases}$$

Secondly, we presume that bit d_i is 0. In that case only the square phase m_{temp}^2 will be executed so we similarly divide messages to subsets M_3 with reduction and M_4 without reduction. Oracle O_2: [3][12]

$$O_2(m) = \begin{cases} 1 & \text{if } m_{temp}^2 \text{ is done with final reduction} \\ 0 & \text{if } m_{temp}^2 \text{ is done without final reduction} \end{cases}$$

We now get 4 subsets of M:

$$M_1 = \{m \in M : O_1(m) = 1\}$$
$$M_2 = \{m \in M : O_1(m) = 0\}$$
$$M_3 = \{m \in M : O_2(m) = 1\}$$
$$M_4 = \{m \in M : O_2(m) = 0\}$$

Let $T_i(M_i)$ be the mean time of computing messages from M_i.

Certainly, only one of oracles is giving us the right results. We can compare time difference between O_1 and O_2. That means if $T_1 - T_2$ is greater than $T_3 - T_4$ then we can be sure that bit d_i is 1, otherwise $d - i$ is 0. The problem from multiply attack is no more actual because one of the differences have to be higher than other. [3][12]

Chapter 5

Defense

5.1 Additional reduction

The most obvious defense is to add dummy subtraction to Montgomery reduction algorithm which does not change any value but consume the same amount of time as if the real subtraction was performed. This should not significantly slow the computation but it totally eliminate this type of timing attack by making Montgomery reduction constant time function.

5.2 Masking

We can mask the ciphertext before computation of $c^d \pmod n$ so the attacker will not know which cipher text is decrypted. It is done simply by generating pair of masks before each exponentiation. We generate random mask m. Then we compute m':

$$m' = (m^{-1})^e \pmod n$$

where e is public exponent.

Before each exponentiation we multiply the ciphertext c with mask m' so we get masked x_m:

$$\begin{aligned} x_m &= (c * m')^d \pmod n \\ &= (c * (m^{-1})^e)^d \pmod n \\ &= c^d * m^{-1} \pmod n \end{aligned} \tag{5.1}$$

from where we can see that c^d is our desired message masked by m^{-1}. Then we simply recover x by multiplying by m:[2]

$$\begin{aligned} x &= x_m * m \pmod{n} \\ &= x * m^{-1} * m \pmod{n} \\ &= x \pmod{n} \end{aligned} \tag{5.2}$$

To avoid situation when even generating of mask could become target of timing attack, there is simple workaround. To generate new mask, just square the mask pair:[2]

$$m = m^2 \pmod{n}$$
$$m' = m'^2 \pmod{n}$$

Chapter 6

Realisation

6.1 RSA implementation

For our purposes we cannot use existing RSA implementation because they commonly have this vulnerability fixed. So it was needed to write own unsecure implementation of RSA cryptosystem. It is still possible use key generation algorithm from OpenSSL because it is not target of our attack. 128-bit key was used. Python 3.6.1 was used and module Crypto for working with keys. After some problem with time measurement I was forced to write core of the algorithm in C. For support big number which RSA operates with I used BIGNUM library from OpenSSL implementation of RSA. This transformation leads to significant speed-up of attack.

6.1.1 Montgomery

The main part of RSA is mechanism for modular exponentiation. As was told before we are using Montgomery multiplication for speed-up computation. It is based on pseudocode in section 2.2.2.

```
def montgomery_product(a, b, n, r, n_inv):
    t = (a * b)
    m = ((t & (r - 1)) * n_inv) & (r - 1)
    u = (t + m * n) >> (r.bit_length() - 1)
    if u > n:
        return u - n
    return u
```

Some optimization was done to let reduction have greater time impact. Instead of modulo r is used bitwise AND with $r - 1$ and instead of division by r is used bitwise shift to right $r.bit\_length() - 1$.

Beacause operations in Python are not constant time so that attack have been failning. Decision was made to transform exponatiation of message to C with using BIGNUM library from OpenSSL which is used in OpenSSL RSA implementation to handle large integers.

```
BIGNUM * mon_prod(BIGNUM * a, BIGNUM * b, BIGNUM * N,
                  int R_length, BIGNUM * Ni, BIGNUM * R){

    BIGNUM * t = BN_new();
    BIGNUM * m = BN_new();
    BIGNUM * u = BN_new();
    BN_mul(t, a, b, ctx);

    BN_mod_mul(m, t, Ni, R ,ctx);

    BN_mul(u, m, N, ctx);

    BN_add(m, u, t);
    BN_rshift(u, m, R_length);

    if (BN_cmp(u, N) >= 0 ){
        BN_sub(m, u, N);
        return m;
    }

    return u;
```

I have ommited alocation and destruction local variables, to make code more readable.

Altough this transformation brings significant speed-up of execution, the successfulness of attack was even worse. Even when I had added some dummy operations in reduction branch, the successfulness was not improved.

6.1.2 Square and Multiply

Due to computation in Montgomery base we also need to little edit the square and multiply algorithm to transform arguments to Montgomery base and at the end back to normal base. We also need precompute r and n^{-1}.

```
def square_and_multiply(ot, n, e):
    r = 2 ** (n.bit_length())
    g, n_inv, r_inv = egcd(n, r)

    if (r * r_inv + n * n_inv) == 1:
        n_inv = -n_inv % r
    else:
        raise Exception("bad GCD")

    ot = (ot * r) % n
    st = (1 * r) % n
    for i in "{0:b}".format(int(e)):
        st = montgomery_product(st, st, n, r, n_inv)
        if i == '1':
            st = montgomery_product(st, ot, n, r, n_inv)
    return montgomery_product(st, 1, n, r, n_inv)
```

As I had tried to eliminate any unnecessary operations I have moved computation of gcd, r^{-1} and n' outside of measured part of code. I would had liked to try if this "cheating" would help to improve successfulness of attack.

After transforming this to C in this function remains only calling Python C-extension , where is whole computation of square and multiply algorith.

```
static PyObject *
montgomery_mult(PyObject *self, PyObject *args)
{
    BIGNUM * N = BN_new();
    BIGNUM * Ni = BN_new();
    BIGNUM * R = BN_new();

    BIGNUM * ot = BN_new();
    BIGNUM * st = BN_new();
    BIGNUM * one = BN_new();
    BN_CTX * ctx = BN_CTX_new();

    char * a, *n, *bit_exp, *sts, *n_inv;
    char *r;
    int length;
    unsigned int i;
    if (!PyArg_ParseTuple(args, "ssssis", &a, &n,  &bit_exp, &r, &length,
        return NULL;
```

```
    BN_dec2bn(&one, "1");
    BN_dec2bn(&N, (const char*) n);
    BN_dec2bn(&Ni,(const char*) n_inv) ;
    BN_dec2bn(&ot,(const char*) a);
    BN_dec2bn(&R,(const char*) r);

    BN_mod_mul(ot, ot, R, N, ctx);

    BN_mod_mul(st, one, R, N, ctx);

    for( i = 0; i < strlen(bit_exp); i++){
        st = mon_prod(st, st, N, length, Ni,R);
        if( bit_exp[i] == '1' ){
            st = mon_prod(st, ot, N, length, Ni, R);
        }
    }

    st = mon_prod(st, one, N, length, Ni, R);

    sts = BN_bn2dec(st);
    return Py_BuildValue("s", sts);
}
```

6.1.3 Encryption and decryption

Encryption and decryption are done just by loading keys from .pem file, then passing them to square_and_multiply function

6.2 Attack implementation

6.2.1 Generating and sorting messages

For both types of attack we are starting with set of randomly generated messages. We give them to oracle which tell us which subset message belongs to. Python module *timeit* is used for time measurements. This chunk of code assign times to messages:

```
import timeit

message_times = dict()
message_range = 50000

for i in range(0, message_range):
    tmp = random.randint(0, n)

    t = timeit.Timer('decrypt.decrypt(int(m1))',
                     setup='import decrypt; m1 = %i' % tmp)

    r = t.timeit(1)

    message_times[tmp] = r
```

6.2.1.1 Multiply

In this version we are attacking multiply operation. We use oracle which is very similar to RSA square_and_multiply function only with one difference. When the final reduction is processed, function return not only result of exponentiation but also bit which tell us that the reduction have been done.

```
...
if u > n:
  return u - n, 1
return u, 0
```

Based on this bit we decide in which subset the message is. The subsets are distinct. Experimentally, we can say that about one quarter of messages belongs to subset with reduction computed.

6.2.1.2 Square

Square attack is similar but we have two oracles which are telling us about reduction on squaring phase. Every time we give the oracle even exponent so multiplication phase will never be the last operation. Each of these oracles divide set of messages to two subsets which are distinct to each other. Each message belongs to one of M_1 or M_2 and to one of M_3 or M_4

6.2.2 Deciding the bit

6.2.2.1 Multiply

We will compare mean times of the subsets of messages. If M_1 is significantly greater then we set guessed bit to 1 and if they differ slightly we set the bit to

0. There is problem with telling what is significant difference because there is lot of noise. The noise is caused by other reductions done by other bits of secret key.

6.2.2.2 Square

We will compare differences between oracles. If oracle predicting multiply has greater difference between subsets we set 1, otherwise we set 0

I tested two different implementation of square attack. The difference is between oracles. One implementation has naive oracle which simply do whole square and multiply algorithm for each message. The second approach is to safe values of particular powers so the oracle does not need to compute whole square and multiply algorithm in each iteration. It just need one square and optional multiplication in each step.

On the other hand, the naive implementation gives better results but is slightly slower.

6.2.3 Assembling secret exponent

After every guessed bit, it is added to variable d which is used by the oracles. After concatenation the new guessed exponent is tested if it is correct exponent. The the test is:

- Pick some message from set
- Decrypt that message
- Do exponentiation of encrypted message to guessed private exponent d

During attack on square we are one cycle ahead so we have no option how to decide LSB so we just try to test both values of last bit.

For better fault toleration we could try guessing of last few bits by brute-force. When the whole attack is finished unsuccessfuly, we can go back few bits and try to all the possibilities of key suffix. It seems reasonable to do this with maximally last 10 bits. There will be extra 1024 encryption and decryption, but it is still more efficient then repeating whole attack.

6.3 Problems and experiments

After attack written in Python fails to guess private key, there were hopeful solution to rewrite the crucial part of RSA to C. I have written Python C-

extension module which handle exponatiation of message. Module is compiled using python setup file:

```
from distutils.core import setup, Extension
setup(name='montgomery', version='1.0',
      ext_modules=[Extension('montgomery',
                             sources=['../montgomery.c'],
                             libraries=['crypto', 'ssl'],
                             )])
```

After builded it is imported like any other Python modul. With this approach I get singificant speed-up. On the other hand, guessing of key got much worse, it can guess maximally 10 bits of key. I was thinking, that it is because the execution time of square and multiply in C is significaly lesser then in Python so that any other operation in Python will have greater impact on final time than single iteration of square and multiply. I have moved precomputation of r^{-1} and n' outside of measured part of code. Unfortunately, it has almost no effect.

Second thought was that reading key from disk is also taking too much time. It could distort measured times by a big amount. But if this will be removed from measured part, the attack will no longer be valid, beacause we use private key which we are guessing as a parameter to time meassuring function. Even thought that change does not help to solve the problem and attack is still failling.

Conclusion

In my environment it was impossible to make any attack sufficient. Attack on multiply have a great problem setting coefficient when the average times of two message groups differs. Which was expectable by nature of this form of attack. Even with sufficient coefficient and 10 000 samples there was no more than 50% success on guessing first unknown bit so it looks completly random

Attacking square was far more interesting. On 50 000 samples algorithm occasionally fails up to guessing 3 bits, but there are more cases when algorithm correctly guess more than 40 bits of key.

To improve performance I have rewriten core of the decrypting algorithm from Python to C language. It did have better performance, but from lesser execution time there are more space for noises to distort measurements.

I have probably identified two biggest noise generators which are precomputation of r^{-1} and n' and reading private key from disk. But when I try to meassure time without these operations it had almost no effect on successfulness of attack.

I have tried lot of minor changes in code, trying to isolate program in docker, running extended Euclidean algorihm in C and lot of another changes but nothing have made any mentionable result.

I am asking question, if it is even possible to make this attack work on my home workstation. Because there is lot of noise, iteruption from system and other events which can distort my measurements.

Because attacks do not work entirely there was no need of implementing defenses in RSA implementation.

Bibliography

[1] http://www.networkworld.com/. Encryption. 2008, [Online; accessed May 14, 2017]. Available from: http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html

[2] Kocher, P. C. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, ISBN 978-3-540-68697-2, pp. 104–113, doi:10.1007/3-540-68697-5_9. Available from: http://dx.doi.org/10.1007/3-540-68697-5_9

[3] Dhem, J.-F.; F.Koeune; et al. Guide to Practical Implementation of the Timing Attack. June 1998, [online], [cit. 2017-05-15]. Available from: http://www.cs.jhu.edu/~fabian/courses/CS600.624/Timing-full.pdf

[4] CONRAD, K. EULER'S THEOREM. 2018, [online], [cit. 2018-05-14]. Available from: http://www.math.uconn.edu/~kconrad/blurbs/ugradnumthy/eulerthm.pdf

[5] Arun-Kumar, S. Algorithmic Number Theory. 2018, [online], [cit. 2018-05-14]. Available from: http://www.cse.iitd.ernet.in/~sak/courses/ant/notes/ant.pdf

[6] Sury, B. Multivariable Chinese Remainder Theorem. 2018, [online], [cit. 2018-05-14]. Available from: https://www.ias.ac.in/article/fulltext/reso/020/03/0206-0216

[7] Kaliski, B. The Mathematics of the RSA Public-Key Cryptosystem. *RSA Laboratories*, 2018, [online], [cit. 2018-05-14]. Available from: http://www.mathaware.org/mam/06/Kaliski.pdf

[8] Lenstra1, A.; Tromer, E.; et al. Factoring estimates for a 1024-bit RSA modulus. 2018, [online], [cit. 2018-05-14]. Available from: https://www.cs.tau.ac.il/~tromer/papers/factorest.pdf

[9] 2018, [online], [cit. 2018-05-14]. Available from: https://web.archive.org/web/20130507091636/http://www.rsa.com/rsalabs/node.asp?id=2092

[10] DI Management:. Using the CRT with RSA. December 2013, [online], [cit. 2017-05-15]. Available from: http://www.di-mgt.com.au/crt_rsa.html

[11] Koç, K. Montgomery Multiplication. *Istanbul Sehir University and University of California Santa Barbara*, 2017, [online], [cit. 2017-05-15]. Available from: http://colinandmargaret.co.uk/Research/Mont_Mult_2ndEd_v4.pdf

[12] BUCEK, J. Utoky postrannimi kanaly. 2013, (lecture), [cit. 2017-03-27], [online]. Available from: https://edux.fit.cvut.cz/courses/MI-KRY/_media/lectures/07/side1.pdf

Appendix A

Acronyms

RSA Rivest, Shamir, Adleman

MSB Most significant bit

LSB Least significant bit

CRT Chinese remainder theorem

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format