



## ASSIGNMENT OF BACHELOR'S THESIS

**Title:** Timing Attack on the RSA Cipher  
**Student:** Martin Andryšek  
**Supervisor:** Ing. Jiří Bůžek  
**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

16th May 2017



---

## **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 16th May 2017

.....

Czech Technical University in Prague  
Faculty of Information Technology

© 2017 Martin Andřýsek. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### **Citation of this thesis**

Andřýsek, Martin. *Timing Attack on the RSA Cipher*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017. 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 čtverců s Montgomeryho násobením. Útok se zaměřuje na měření času trvání dešifrování rozdílných zpráv s určitými vlastnostmi. Práce popisuje základní principy a slabiny RSA kryptosystému. Výsledkem práce je demonstrativní aplikace, která bude použita ve vyuce předmetch, zabývajících 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

<b>Introduction</b>	<b>1</b>
<b>1 State-of-the-art</b>	<b>3</b>
<b>2 RSA</b>	<b>5</b>
2.1 Principle . . . . .	5
2.2 Optimization . . . . .	6
<b>3 Attacks</b>	<b>11</b>
3.1 Attack on multiply . . . . .	11
3.2 Attack on square . . . . .	12
<b>4 Defense</b>	<b>15</b>
4.1 Additional reduction . . . . .	15
4.2 Masking . . . . .	15
<b>5 Realisation</b>	<b>17</b>
5.1 RSA implementation . . . . .	17
5.2 Attack implementation . . . . .	18
<b>Conclusion</b>	<b>21</b>
<b>Bibliography</b>	<b>23</b>
<b>A Acronyms</b>	<b>25</b>
<b>B Contents of enclosed CD</b>	<b>27</b>



---

## List of Figures



---

# Introduction

In 1996 Paul Kocher presented timing attack on several cryptosystems including RSA. The cryptosystems have in common that all of them are using modular exponentiation and 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.

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.





## State-of-the-art

By exploiting data dependency on secret exponent we should be able to recover key.

We should decide what is the optimal amount of messages. It should not be too many so that we can guess the key in reasonable time but it have to be enough to guess the key.

Next, we should create some defensive mechanism in cryptosystem in order to make attack fail.



---

# RSA

RSA is public-key cryptosystem which was invented by Ron Rivest, Adi Shamir and Leonard Adleman. The cryptosystem was published in the 1977.

## 2.1 Principle

The cipher is based on modular exponentiation. The whole process of crypting message is divided to four steps

### 2.1.1 Key generation

This is steps needed to generate public and private keypair

- Generate  $p$  and  $q$ , which have to be distinct prime numbers.
- Compute  $n$ , where  $n = pq$
- Compute Euler's totient function  $\phi(n)$ . Because we know  $p$  and  $q$  it is simple to compute it.

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

- Generate  $e$  such as  $\gcd(e, \phi(n)) = 1$
- Compute  $d = e^{-1} \bmod \phi(n)$
- The pair  $(e, n)$  is released as public key
- The pair  $(d, n)$  is secret private key

## 2. RSA

---

### 2.1.2 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.

### 2.1.3 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.

### 2.1.4 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.

$$|c^d|_n = |(m^e)^d|_n = |m^{ed}|_n = |m^1|_n = m$$

### 2.1.5 Signing

RSA signing is used to verify identity of sender. The process is really similar to decryption but instead of message we are powering hash of the message. We are using our private exponent to sign hash so the receiver could easily decrypt hash using public exponent and compare it to actual hash of the received message.

## 2.2 Optimization

Because we generally use high value of modulus  $n$  the exponentiation of such high numbers is very time consuming so there are some algorithms to increase speed of computation

### 2.2.1 Chinese remainder theorem

By using CRT we can significantly speed up decryption of received messages. This method is not usable during encrypting phase because we need to know  $p$  and  $q$  factors of  $n$ . Assuming that  $p > q$  we can precompute:

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

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

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

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

### 2.2.2 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 greater power of 2 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

$$\begin{aligned} \bar{u} &= \bar{a}\bar{b}r^{-1} \pmod{n} \\ &= (ar)(br)r^{-1} \pmod{n} \\ &= (ab)r \pmod{n} \end{aligned} \tag{2.1}$$

## 2. RSA

---

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:    $\bar{u} \leftarrow MonRed(\bar{a}, \bar{b})$ 
7:    $u \leftarrow MonRed(\bar{u}, 1)$ 
8:   return  $u$ 
9: end function
```

---

### 2.2.3 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 theses this Square and Multiply algorithm is called Montgomery exponentiation

---

**Algorithm 3** Square & Multiply algorithm

---

```
1: function SQUARE_AND_MULTIPLY( $m, e, n$ )
2:    $c \leftarrow 1$ 
3:    $k \leftarrow \text{BitLen}(e)$ 
4:   for  $i \leftarrow k - 1, 0$  do
5:      $c \leftarrow \text{MonMult}(c, c)$ 
6:     if  $e[i] == 1$  then  $\triangleright$   $i$ th bit of exponent  $e$ 
7:        $c \leftarrow \text{MonMult}(c, m)$ 
8:     end if
9:   end for
10:  return  $c$ 
11: end function
```

---





# Attacks

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

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.

## 3.1 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  $i$ th bit. Let  $d = d_1, d_2, \dots, d_k$  where  $k$  is bit length of  $d$  and  $d_1$  is MSB. We can assume that  $d_1 = 1$  so we can attack bit  $d_2$ .

We need oracle  $O$  which predict whether final Montgomery reduction happened during multiply step:

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

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

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

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

### 3. ATTACKS

---

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.

### 3.2 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  $i$ th bit we compute  $m_{temp}$  which has value before unknown possible multiplication step.[1]

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:

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

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  $O_2$ :

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

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.



# Defense

## 4.1 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.

## 4.2 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  $x_m$ :

$$\begin{aligned} x_m &= (c * m')^d \pmod n \\ &= (c * (m^{-1})^e)^d \pmod n \\ &= c^d * m^{-1} \pmod n \end{aligned} \tag{4.1}$$

#### 4. DEFENSE

---

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

$$\begin{aligned} x &= x_m * m \pmod{n} \\ &= x * m^{-1} * m \pmod{n} \\ &= x \pmod{n} \end{aligned} \tag{4.2}$$

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]

$$\begin{aligned} m &= m^2 \pmod{n} \\ m' &= m'^2 \pmod{n} \end{aligned}$$

---

## Realisation

### 5.1 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.

#### 5.1.1 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  $r.bit\_length() - 1$ .

#### 5.1.2 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)
```

### 5.1.3 Encryption and decryption

Encryption and decryption are done just by loading keys from .pem file, then passing them to square\_and\_multiply function

## 5.2 Attack implementation

### 5.2.1 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.

#### 5.2.1.1 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.

### 5.2.1.2 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  $M_4$

## 5.2.2 Deciding the bit

### 5.2.2.1 Multiply

### 5.2.2.2 Square

## 5.2.3 Assembling secret exponent

**5.2.3.0.1** 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
- encrypt that message
- power encrypted message to guessed private exponent  $d$



---

## Conclusion



---

## Bibliography

- [1] Dhem, J.-F. Guide to Practical Implementation of of the Timing Attack.
- [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](http://dx.doi.org/10.1007/3-540-68697-5_9)



## Acronyms

**RSA** Rivest, Shamir, Adleman

**MSB** Most significant bit

**LSB** Least significant bit

**CRT** Chinese remainder theorem





## 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 $\text{\LaTeX}$ 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