TIMING TAGS IN EXPONENTIATIONS FOR RSA

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

October, 1999

By
Brice Canvel
Department of Mathematics

Contents

Abstract **6**

Declaration **7**

Acknowledgements **8**

1 Introduction **9**

2 RSA algorithm **12**

2.1 Public key cryptography 12

2.2 Some useful mathematics 13

2.2.1 The Euclidean algorithm 13

2.2.2 Congruences 14

2.3 The algorithm 16

3 Coding the RSA algorithm **18**

3.1 Mathematica and RSA 18

3.2 RSA and C++ 20

3.2.1 The LargeNumber class 21

3.2.2 Greatest Common Divisor and the Euclidean algorithm . . 22

3.2.3 Head's algorithm, modular multiplication and modular exponentiation 23

3.2.4 Finding Primes 25

3.2.5 Timings in C++ 26

3.3 Outline of main program 26

4 Timing Attacks and DPA **28**

4.1 Introduction 28

4.2 Timing attack 29

4.3 Power Analysis . . . 32
4.4 Possible Countermeasures . . . 35
4.4.1 Timing attack . . . 35
4.4.2 DPA . . . 35

5 Timing analysis **37**
5.1 The data . . . 37
5.2 Analysing the data . . . 39
5.3 Conclusion . . . 42

6 Conclusion and further work **45**
6.1 Conclusion . . . 45
6.2 Further work . . . 47

Bibliography **48**

A Code **51**
A.1 prog.cc . . . 52
A.2 largenumber.h . . . 57
A.3 largenumber.cc . . . 59
A.4 timer.h . . . 74

B Graphs **77**

List of Tables

3.1 RSA numbers obtained using Mathematica 21

5.1 First entry in the file data.dat 38
5.2 First entry in the file check.dat 38
5.3 First entry in the file timing.dat 39

List of Figures

4.1 Electronic trace of six operations done by a smartcard [12] 33
4.2 Electronic trace of part of an encryption operation [12] 34

5.1 Timings for an encrypting key of size 20 bits with p and q of size 50 bits 40
5.2 Timings for an encrypting key of size 20 bits with p and q of size 65 bits 41
5.3 Timings for an encrypting key of size 20 bits with p and q of size 195 bits 43

B.1 Timings for an encrypting key of size 20 bits with the size of p and q varying from 50 to 75 bit 78
B.2 Timings for an encrypting key of size 20 bit with the size of p and q varying from 80 to 105 bit 79
B.3 Timings for an encrypting key of size 20 bit with the size of p and q varying from 110 to 135 bit 80
B.4 Timings for an encrypting key of size 20 bit with the size of p and q varying from 140 to 165 bit 81
B.5 Timings for an encrypting key of size 20 bit with the size of p and q varying from 170 to 195 bit 82
B.6 Timings for an encrypting key of size 20 bit with the size of p and q varying from 200 to 225 bit 83
B.7 Timings for an encrypting key of size 20 bit with the size of p and q varying from 230 to 255 bit 84
B.8 Timings for an encrypting key of size 20 bit with the size of p and q varying from 260 to 285 bit 85
B.9 Timings for an encrypting key of size 20 bit with p and q of size 290 bit 86

B.10 Timings for an encrypting key of size 60 bit with the size of p and q varying from 50 to 65 bit . . . 87
B.11 Timings for an encrypting key of size 60 bit with the size of p and q varying from 70 to 85 bit . . . 88
B.12 Timings for an encrypting key of size 60 bit with the size of p and q varying from 90 to 105 bit . . . 89
B.13 Timings for an encrypting key of size 60 bit with the size of p and q varying from 110 to 125 bit . . . 90
B.14 Timings for an encrypting key of size 60 bit with the size of p and q varying from 130 to 145 bit . . . 91
B.15 Timings for an encrypting key of size 60 bit with the size of p and q varying from 150 to 165 bit . . . 92
B.16 Timings for an encrypting key of size 60 bit with the size of p and q varying from 170 to 185 bit . . . 93
B.17 Timings for an encrypting key of size 60 bit with the size of p and q varying from 190 to 205 bit . . . 94
B.18 Timings for an encrypting key of size 60 bit with the size of p and q varying from 210 to 225 bit . . . 95
B.19 Timings for an encrypting key of size 60 bit with the size of p and q varying from 230 to 245 bit . . . 96
B.20 Timings for an encrypting key of size 60 bit with the size of p and q varying from 250 to 265 bit . . . 97
B.21 Timings for an encrypting key of size 60 bit with the size of p and q varying from 270 to 285 bit . . . 98
B.22 Timings for an encrypting key of size 60 bit with p and q of size 290 bit . . . 99
B.23 Correlation between the size of p and q (from 50 to 105 bit) and the digits in the exponent with encrypting key of length 20 bit . . . 100
B.24 Correlation between the size of p and q (from 50 to 105 bit) and the digits in the exponent with encrypting key of length 60 bit . . . 101
B.25 Correlation between the size of p and q (from 50 to 290 bit) and the digits in the exponent with encrypting key of length 20 bit . . . 102

To my mum, my grand-mothers, my sister Céline and my brother Cyril.

Abstract

Encryption and decryption procedures for electronic communications depending on public key methods, such as RSA, rely for their security on the difficulty of factoring a number n that is a product of two large primes p, q. Nevertheless, public key encryption procedures have been shown to be vulnerable to attack through analysis of power consumption recordings and timing intervals between operations. The RSA algorithm involves modular arithmetic of exponentiation with large exponents. This dissertation has been designed to illustrate the effect of correlations between timing intervals and bits being processed by placing flags in a C++ implementation of RSA using the Head algorithm for exponentiation. Special C++ procedures were needed to handle large integers. Using the code that was developed, data in the form of timing traces was collected for encryption of a fixed plaintext using an increasing sequence of size of p and q from 50 to 290 bits and varying the size of the public key from 20 to 100 bits. As p and q increase in size we see that the timing traces more clearly reveal the sequence of binary digits in the exponent. A potential attacker with access to even noisy versions of such timing traces, obtained for example by electronic probes, would have helpful information for an iterative statistical procedure to find the secret key.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Acknowledgements

I would like thank my supervisor Prof Kit Dodson whose attitude, patience and guidance has made this year worthwhile. I would also like to take this opportunity to thank the Mathematics department at UMIST for their encouragement throughout my university life.

Chapter 1

Introduction

In today's society, information has taken a different form. Typewriters and handwriting are now rarely used. This is due to the appearance of computers that have become more and more powerful and more user-friendly. Nowadays, every secretary has a computer on the desk and is trained to use word processing packages and spreadsheet which create better documents with colours, graphics and tables allowing a better presentation and analysis of the data, an output pleasing to the eye, in other words, a document that one will want to read.

This new digital information has become more and more present as networks have become faster. We are now able to analyse and present data that can be transferred to someone a long way away within minutes. This is very true with the expansion of the Internet which originally was only used by scientists. Today, all Universities and major companies use this worldwide network to transfer their data, many households have a personal computer and use it to send emails which can contain text, images, sounds and even moving pictures. Banks are all linked together which makes the transfer of money very fast and efficient and the fight against fraud an easier one. Indeed, when someone uses their credit card in a shop, it is possible for the owner to automatically check whether the customer is in credit or not.

The Internet has become so popular that many things have changed or are changing in everyday life. For example, it is now possible to book plane or train tickets without having to wait in very long queues. Most things can be purchased on-line and paid the same way using a credit card.

We even carry this digital information with us through the use of smartcards, credit-sized devices that contain a microchip. On this chip, a lot of information can be contained such as the medical records of a patient or some information relating a customer to its bank making it possible to identify the authenticity of the information provided.

All this transfer of information from bank to bank, from customer to shop owner, from friend to friend, etc, needs to be protected in some way. Indeed, when one gives a credit card number on the Internet, no one but the authorised receiver of this number should be able to identify what it is. If one could obtain the card number and other information being transmitted, then he/she could easily make use of it to purchase goods as well. New devices are now coming out which are like automated teller machine in reduced form that can be linked to a computer in order to make electronic payments over the Internet by using the signature process contained in the new credit cards.

Mainly, the protection of information is done through the use of what is known as data encryption. Data encryption is the use of mathematical algorithms applied to some data which as a result makes this data unclear to someone who is not allowed to see it. Many algorithms have been developped over the years, some very old, some more recent. These encrypting techniques are now at the front of the scene with this need for information protection. One type of cryptosystem is known as public key cryptography and RSA is one example of this. Basically, it uses two keys, one to encode the data and one to decode it. One of the keys, the decoding one or secret key, is kept secret so that only its owner can use it and the other key is made public so that everyone uses it to send protected information to the owner of the key. This method is described in chapter 2.

Even though the information is protected, there are people, malicious people, who will try to breach this security in order to make illegal use of it. The RSA algorithm talked about is one of the most secure and widely used algorithms to encode information. It still has some security weaknesses that one should be aware of. For that reason, scientists around the world try to find new ways to break the algorithm and give countermeasures to these risks. It has been shown that many security weaknesses of RSA are due to misuse of the algorithm and hence, someone aware of those countermeasures will be able to implement them when designing a system. By being ahead of hackers, one can rely on the protection of the information. As far as the RSA is concerned, new techniques involving the analysis of the power consumption of a device and timing intervals between operations have been developped in order to break the implementation of the algorithm. These techniques are studied in chapter 4.

In this dissertation, we have implemented a version of RSA in C++ using special procedures to handle very large integers. The implementation is shown in detail in chapter 3. We then used this code to obtain some timing traces for encryption and decryption of a fixed plaintext message, varying the size of p and q from 50 to 290 bits and the size of the encrypting key from 20 to 100 bits. We then used these results to show the correlation between these timing traces and the bits of the encrypting key being processed. Similar results can be obtained for the decrypting key. We show in particular how the traces become clearer and clearer as p and q increase in size. The results obtained are analysed in chapter 5. With noisy versions of these traces such as an electronic probe of hte power cunsumption) and the use of statistical tools, the security of a system could be in great danger as one could use these to find the secret. Similar attacks against devices such as smartcard have already been successfully tested. [11]

Chapter 2

RSA algorithm

2.1 Public key cryptography

There are two different types of cryptographic systems, namely symmetric and asymetric. In the case of a symmetric system, anyone who can encrypt a message is also able to decrypt it and the sender and the receiver have a common secret key which they both must know before starting encryption. This is not a very secure way of encrypting messages as the secret key could easily be intercepted by a third party.

For an asymetric system, the first property in the definition of a symmetric system is no longer valid. In this process, the enciphering key is different from the deciphering key. It is also called public key cryptography as the encrypting key is made public so that everyone can use it. However, the decrypting key is kept secret by the publisher of the encrypting key. To each enciphering key corresponds a unique deciphering key. Therefore, anyone can encrypt a message but only the owner of the secret key and corresponding public key can decrypt it.

In a more graphic way, public key cryptography can be schematised as follows. Let us call E the enciphering (or public) key and D the deciphering (or secret) key. Denote by $E(m)$ the result obtained from encrypting a message m using the public key E and $D(m)$ the result obtained from decrypting a ciphertext m using the secret key D. Then, we obtain the following expression from the definition for public key cryptography,

$$D(E(m)) = m. \tag{2.1}$$

The RSA algorithm, which we study in more detail later is an example of public key cryptography.

2.2 Some useful mathematics

We now give some definitions and theorems needed in the understanding and proof of the RSA algorithm, namely the Euclidean algorithm and congruences.

2.2.1 The Euclidean algorithm

The Euclidean algorithm was derived by the mathematician Euclid. It finds the greatest common divisor of two positive integers. This is used in practice to create the public and private keys in the RSA algorithm studied in section 2.3.

Let us denote the greatest common divisor of a and b by (a, b) where a and b are positive integers.

The following theorem is used in proving the Euclidean algorithm.

Theorem 2.2.1 *If r is the remainder of a divided by b then $(a, b) = (b, r)$.*

The Euclidean algorithm is set up as follows,

Algorithm 2.2.1 (Euclidean algorithm) *Let $r_0 = a$ and $r_1 = b$ be integers such that $a \geq b > 0$. Let $r_j = r_{j+1}q_{j+1} + r_{j+2}$ with $0 < r_{j+2} < r_{j+1}$ for $j = 0, 1, 2, \ldots, n-2$ and $r_{n+1} = 0$ then $(a, b) = r_n$, the last non-zero remainder.*

A proof for theorem 2.2.1 and and algorithm 2.2.1 is given in [1] p. 81.

Here is an example on how the Euclidean algorithm can be used to find the greatest common divisor of two integers.

Example 2.2.1 *Find (533,117).*
Using the set of equation above, we have
533=4.117+65
117=1.65+52
65=1.52+13
52=4.13.
Hence (533,117)=13.

2.2.2 Congruences

Congruences are at the heart of the RSA algorithm and also very useful in number theory generally. The theory of congruences was developed by Gauss at the start of the 19th century.

Definition

First, we define the notion of divisibility.

Definition 2.2.1 *Let a and b be two positive integers. Then we say that a divides b if there is an integer c such that $b = ac$. ([1] p. 36)*
If a divides b, we write $a|b$ and if a does not divide b, we write $a \not| b$.

Using definition 2.2.1, we define congruences as follows.

Definition 2.2.2 *Let m be a positive integer. If a and b are positive integers, we say that a is congruent to b modulo m if $m|(a - b)$. ([1] p. 120)*
If a is congruent to b modulo m, we write $a \equiv b \mod m$. If $a \not| b$, we write $a \not\equiv b \mod m$.

For use in proofs and ease of understanding of congruences, it is possible to translate them into equalities using the following theorem,

Theorem 2.2.2 *If a and b are integers, then $a \equiv b \mod m$ if and only if there is an integer k such that $a = b + km$. ([1] p. 120)*

We will require to do some arithmetic with congruences as we do with equalities and many of the properties of the latter also hold for congruences. The following rules apply to addition, subtraction and multiplication.

Theorem 2.2.3 *If a, b, c, d and m are integers such that $m > 0$, $a \equiv b \mod m$ and $c \equiv d \mod m$, then*

1. $a + b \equiv b + d \mod m$
2. $a - c \equiv b - d \mod m$
3. $ac \equiv bd \mod m$.

([1] p. 123)

The following theorem concerning multiplication can also be useful.

Theorem 2.2.4 *If* $a \equiv b \mod m_1$, $a \equiv b \mod m_2$,...,$a \equiv b \mod m_k$ *and* m_1, m_2,..., m_k *are pairwise relatively prime that is* $(m_i, m_j) = 1$ *for* $i, j = 1, \ldots, k$, *then* $a \equiv b \mod m_1 m_2 \ldots m_k$. *([1] pp. 124-125)*

Linear congruences, Fermat's Little Theorem and Euler's Phi function

One useful notion in congruences used to find public and private keys in the RSA algorithm is linear congruences.

Definition 2.2.3 *A congruence of the form* $ax \equiv b \mod m$, *where* x *is an unknown integer is called a linear congruence in one variable. ([1] p. 131)*

The following theorem tells us more about the existence (or non-existence) and number of solutions of a linear congruence in one variable.

Theorem 2.2.5 *Let a,b and m be integers with* $m > 0$ *and* $(a, m) = d$. *If* $d \not| b$, *then* $ax \equiv b \mod m$ *has no solutions. If* $d|b$, *then* $ax \equiv b \mod m$ *has exactly d incongruent solutions modulo m. ([1] p. 131)*

If we now consider congruences of the special form $ax \equiv 1 \mod m$, then there is a solution if and only if $(a, m) = 1$, and then all solutions are congruent modulo m and are called inverse of a modulo m.

One very important theorem used in proving the RSA algorithm is Fermat's LittleTheorem which tells us how to work with certain congruences involving exponents when the modulus is prime.

Theorem 2.2.6 (Fermat's Little theorem) *If* p *is prime and* a *is a positive integer with* $p \not| a$, *then* $a^{p-1} \equiv 1 \mod p$. *([1] p.187)*

This theorem can be generalised using Euler's phi-function.

Definition 2.2.4 (Euler phi-function) *Let* n *be a positive integer. The Euler phi-function* $\phi(n)$ *is defined to be the number of non negative integers* b *less than* n *which are prime to* n *:*

$$\phi(n) = \{b|0 \leq b < n, (b, n) = 1\}.$$

([3] p. 15)

Example 2.2.2 *Calculate the Euler phi-function for a positive prime n.*
For every positive integer k such that $0 \leq k < n$, we have $(n, k) = 1$. Hence

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

for n prime.

The Euler phi-function is multiplicative that is $\phi(mn) = \phi(m)\phi(n)$ whenever $(m, n) = 1$. ([3] p. 21) This leads to the extension of Fermat's Little theorem which is at the heart of the RSA algorithm.

Theorem 2.2.7 (Euler's theorem) *Let a and m be any positive integers such that $(a, m) = 1$. Then $a^{\phi(m)} \equiv 1 \mod m$. ([3] p. 22)*

2.3 The algorithm

As we have said previously, the RSA algorithm is an example of a public key cryptosystem. It was discovered in 1977 by Ronald Rivest, Adi Shamir and Leonard Adleman, hence its name formed the initials of their family names. This algorithm uses very simple ideas in number theory which are combined together in such a clever way that it is a very secure cryptosystem and one of the most famous ones.

The algorithm is as follows,

Algorithm 2.3.1 (The RSA algorithm) *We choose two extremely large prime numbers p and q and we calculate their product n.*
Then, we calculate $\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$ using the multiplicity property of the phi-function and example 2.2.2.
We choose an integer e between 1 such $\phi(n)$ so that $(e, \phi(n)) = 1$.
From theorem 2.2.5, there exists d such that $ed \equiv 1 \mod \phi(n)$. It is possible to find d using the Euclidean algorithm (algorithm 2.2.1).
We then have a public key e and a private key d which is the basis of our public key cryptosystem.
We have a cleartext message M consisting of a sequence of positive integers m_i such that $(m_i, n) = 1$. The encoded message C is then obtained by computing $c_i \equiv m_i^e \mod n$ for all m_i in the cleartext message. This gives us a ciphertext message C composed of the positive integers c_i.

The decoding procedure is similar but in this case, the private key d is used instead of the public key. In other words, we obtain the cleartext message D=M by computing $d_i \equiv c_i^d \mod n = m_i$ for all c_i in the cyphertext message.

The proof for the RSA algorithm is fairly simple and uses the mathematical tools defined in 2.2.

Proof 2.3.1 (The RSA Algorithm) *Let p, q, n, e, d, $\phi(n)$ and M be defined as in algorithm 2.3.1.*
Let m be any element m_i of M.
Then, $(m^e)^d \equiv (m^d)^e \equiv m^{ed} \mod n$. Thus, using the equality form of congruences and the fact that $ed \equiv 1 \mod \phi(n)$, we know that there exists an integer k such that $m^{ed} \equiv m^{k\phi(n)+1} \mod n$.
If $(m,p) \neq 1$ then $m \equiv 0 \mod p \quad \Rightarrow \quad m^{ed} \equiv m \equiv 0 \mod p$.
Since $\phi(p) = p - 1$, we get :
$m^{ed} \equiv m^{k\phi(n)+1} \mod p$
$\Rightarrow \quad m^{ed} \equiv mm^{k\phi(n)} \mod p$
$\Rightarrow \quad m^{ed} \equiv mm^{k(p-1)(q-1)} \mod p$ *from example 2.2.2*
$\Rightarrow \quad m^{ed} \equiv m(m^{p-1})^{k(q-1)} \mod p$
$\Rightarrow \quad m^{ed} \equiv m1^{k(q-1)} \mod p$ *using theorem 2.2.6*
$\Rightarrow \quad m^{ed} \equiv m1 \equiv m \mod p$.
Hence, $m^{ed} \equiv m \mod p$ for any positive integer m. Similarly, $m^{ed} \equiv m \mod q$. Therefore p and q divide $m^{ed} - m$ and since they are both prime, we deduce that $n = pq$ is a divisor of $m^{ed} - m$ so that $m^{ed} \equiv m \mod n\Box$

In the next chapter, we see how this algorithm can be implementation to run on a computer using the C++ language.

Chapter 3

Coding the RSA algorithm

In order to understand the RSA algorithm and obtain results which we know are consistent, we use the mathematical package Mathematica. This has functions we need to implement the RSA algorithm and can be applied using large integers. After having obtained some data, we see how the RSA algorithm can be coded in C++ in order to work with very large moduli and exponents.

3.1 Mathematica and RSA

Mathematica has efficient functions to perform modular arithmetic of large powers of large integers. These functions are included in a package which we load as follows,

```
<<NumberTheory`NumberTheoryFunctions`
<<NumberTheory`PrimeQ`.
```

Then, the following functions are available,

1. `GCD[a,b]`

   gives the greatest common divisor of a and b,

2. `ExtendedGCD[a,b]`

   gives the greatest common divisor of a and b together with numbers x and y in the following form

   ```
   {gcd,{x,y}}
   ```

so that $gcd = xa + yb$,

3. `Mod[m,n]`

   gives the remainder of m divided by n,

4. `PowerMod[a,b,n]`

   gives $a^b \mod n$,

5. `NextPrime[n]`

   gives the smallest prime p such that $p > n$,

6. `PrimeQ[a]`

   gives True if a is a prime number,

7. `EulerPhi[n]`

   gives $\phi(n)$ (see definition 2.2.4).

We can use this functions as follows to see how they are useful in obtaining some results using RSA.

Example 3.1.1 *Firstly, we find two primes p and q :*

```
{p,q}={NextPrime[5583],NextPrime[6195])
```

This gives us

```
{5591,6197}.
```

We then calculate

```
n=pq
34647427.
```

We choose e prime :

```
e=2367921
```

We then check that $(e, \phi(n)) = 1$ *:*

```
ExtendedGCD[EulerPhi[n],e]
{1,{-77402,1132161}}.
```

This also allows us to find the decrypting key d :

```
d=Mod[1132161,EulerPhi[n]]
1132161.
```

We check that $ed \equiv 1 \mod n$ *:*

```
Mod[e*d,EulerPhi[n]]==1
True.
```

We can then encrypt and decrypt a message (say) 1234 *using* e*,* d *and* n *as follows,*

```
PowerMod[1234,e,n]
9488426
PowerMod[9488426,d,n]
1234.
```

This is the result we were expecting.

Using the same process, we obtain the results shown in the table below which we will use later on to check the validity of our implementation of RSA in C++.

3.2 RSA and C++

When coding the RSA algorithm in C++, the biggest problem arising is the limit imposed by the compiler on how big the integers we use can be. As a matter of fact, we can only use integers up to a maximum of perhaps about two billion, depending on the machine we are running the code on. Therefore, it is necessary to implement a way of representing very large integers which are required for good use of the RSA algorithm. This is done creating a class called LargeNumber.

Using this class, we implement code to find the greatest common divisor, the Euclidean algorithm to find the modular inverse of a number, modular multiplication using Head's algorithm and modular exponentiation using the square-and-multiply method. In this section, we describe the algorithms used to produce the C++ code implementation of RSA.

Message	12345678
n	8149951343344931
e	554639
d	4429258993324799
Encrypted message	5032299039605580
n	45986337785906126343912382779695273
e	877467387997
d	12048967505047791221657508443046073
Encrypted message	39421049732029167562364013322838984
n	23895426340894949213080149944927518369454687
e	887611244477
d	8875864207953498393313127573010340256152493
Encrypted message	18026528890676342019492544365436448611682651
n	10169668265654444015760778247386383686770394022600427131499
e	554654888684761
d	12825012848682290015706953686956826899645038027865265177
Encrypted message	75803189588449472183780273674806639744466185899604287759
n	65265361568342331617812017298201639447489792918057565217190861159
e	123554654888684689
d	5359600161104121102944927850567285579942774496104131229815157209
Encrypted message	26512841915726721454826389287649796885153920970173451144432642002

Table 3.1: RSA numbers obtained using Mathematica

3.2.1 The LargeNumber class

The LargeNumber class is a full C++ implementation of a module that allows us to use very large integers in C++. As mentioned before, C++ can only deal with integers that are relatively small compared to the size of numbers we require when using the RSA algorithm. It was therefore necessary, in order to implement RSA in C++, to find a way to represent such large numbers. Considering the length of time given to complete this dissertation, I decided to use a piece of code written by Matthew Caryl that is a very simple implementation of large integers in C++. It provides us with all the necessary operations one wishes to use when dealing with numbers in all four arithmetic operations, comparison operators and input and output operators.

It was necessary to add a few functions to it that are required to do some of the RSA operations such as an implementation of the Euclidean algorithm to find the inverse modulus of a number, modular multiplication and modular exponentiation, and a function to find the greatest common divisor between two numbers. Algorithms for all these added facilities are described in the following sections. It was also necessary to modify the divide function as it did not work properly for very large number.

The integers in the LargeNumber class are represented as binary numbers in the form of an array of zeros and ones and are signed integers so that we can have both positive and negative numbers. Useful functions such as shift left and right are provided which allow fast division and multiplication on the binary numbers.

Now we look at the algorithms that have been implemented in C++ in order to give us the necessary tools to code RSA.

3.2.2 Greatest Common Divisor and the Euclidean algorithm

The greatest common divisor of two integers x and y is computed by using a trial division method. It comes into the RSA algorithm to check the coprimality of the public key e and the Euler-phi function.

Algorithm 3.2.1 (Greatest Common Divisor)

```
While True
   If y=0
      gcd=x and Exit
   End If
   x=Remainder of x/y
   If x=0
      gcd=y and Exit
   End If
   y=Remainder of y/x
End While
```

Then, we have $gcd = (x, y)$.

We use the Euclidean algorithm to calculate the inverse of a modulo m which is needed in finding the private key in the RSA algorithm. The algorithm works as follows,

Algorithm 3.2.2 (Inverse modulus)

```
b=m
c=a
i=0
j=1
```

```
While c>0
   x=b/c
   y=b-x/c
   b=c
   c=y
   y=j
   j=i-jx
   i=y
End While

If i<0
   i=i+m
End If
```

We then have $ia \equiv 1 \mod m$.

3.2.3 Head's algorithm, modular multiplication and modular exponentiation

Head's algorithm is a method for calculating the modular product of two integers. Normally, we would need to handle numbers of size m^2 when multiplying integers x and y $(0 \leq x, y < m)$ modulo m but Head's algorithm does the multiplication without introducing numbers bigger than $4m$ which makes the multiplication process faster. Before showing the algorithm, we need to introduce some notation and a couple of theorems.

Definition 3.2.1 *The integer part of a real number x, denoted $[x]$, is the greatest integer less than or equal to x. ([4] p. 14)*

The following two theorems are used in proving Head's algorithm,

Theorem 3.2.1 *Let $m \geq 2$, $T = [\sqrt{m} + \frac{1}{2}]$ and $t = T^2 - m$. Then $-T \leq t \leq T$ and $T^2 \leq 2m$. ([4] p. 94)*

Theorem 3.2.2 *Let $x = aT + b$ where $0 \leq b \leq T$. Then $0 \leq a \leq T$. Let $y = cT + d$ where $0 \leq c \leq T$. Then $0 \leq c \leq T$. ([4] p. 94)*

Then, Head's algorithm is as follows,

Algorithm 3.2.3 (Head's algorithm) *Assume the same notation as Theorems 3.2.1 and 3.2.2. Define* $z \equiv ad + bc \mod m$*, with* $0 \leq z < m$*.*
Then $ad \leq m$*,* $bc \leq m$ *and* $xy \equiv act + zT + bd \mod m$*.*
Now let $ac = eT + f$*, with* $0 \leq f < T$*.*
Then $0 \leq e \leq T$ *and* $xy \equiv (et + z)T + ft + bd \mod m$*.*
Finally, let $v \equiv z + et \mod m$*, with* $0 \leq v < m$*. Let* $v = gT + h$*, with* $0 \leq h < T$*.*
Then $0 \leq g \leq T$ *and* $xy \equiv hT + (f + g)t + bd \mod m$*. ([4] p. 94)*

Proof 3.2.1 (Head's algorithm) *From theorem 3.2.2, we have that*
$x = aT + b$ *where* $0 \leq b \leq T$*,* $0 \leq a \leq T$*, and*
$y = cT + d$ *where* $0 \leq c \leq T$*,* $0 \leq c \leq T$*.*
Define $z \equiv ad + bc \mod m$*, with* $0 \leq z < m$*.*
Then,
$xy = (aT + b)(cT + d)$
$\Rightarrow \quad xy = acT^2 + bcT + adT + bd.$
Replacing T^2 *by* $t - m$ *we obtain,*
$xy = ac(t - m) + (bc + da)T + bd.$
Now using z *as defined above, we have,*
$xy \equiv act + zT + bd \mod m.$

Now let $ac = eT + f$*, where* $0 \leq f \leq T$*. Suppose that we have* $e > T$*. Then,* $ac \geq et \geq T^2$*, which is false from theorem 3.2.2. Hence,*
$0 \leq e \leq T$*.*
Now replacing ac *by* $eT + f$ *in* $xy \equiv act + zT + bd \mod m$*, we obtain,*
$xy \equiv (et + z)T + ft + bd \mod m.$

Finally, let $v \equiv z + et \mod m$*, with* $0 \leq v < m$*. Let* $v = gT + h$*, with* $0 \leq h < T$*. By theorem 3.2.1, if* $g > T$*, then* $gT + h \geq gT \geq (T + 1)T \geq m$ *which is a contradiction. Hence* $0 \leq g \leq T$*.*
Replacing $z + et$ *by* v *and* v *by* $gT + h$ *in* $xy \equiv (et + z)T + ft + bd \mod m$*, we obtain that*
$xy \equiv hT + (f + g)t + bd \mod m$ *which is the result we wanted to prove* □

Using Head's algorithm and the square-and-multiply method, we are able to compute the modular exponentiation $x^n \mod m$ where $n = d_k d_{k-1} \ldots d_1 d_0$ is the exponent in binary form using the following algorithm,

Algorithm 3.2.4 (Modular exponentiation)

```
Let r=1
While n>0
   Let d=n-2[n/2]
   If d=1 then
      r=xr mod m (using Head's algorithm)
   End If
   x=x^2 mod m  (using Head's algorithm)
   n=(n-d)/2
End While
```

We then have $r \equiv x^n \mod m$*. ([4] p. 90)*

3.2.4 Finding Primes

One tool at the heart of the RSA algorithm is the need for large prime numbers. Indeed, the modulus is composed of two prime numbers and the public and private keys are also to be prime. In order to generate prime numbers, we will use an algorithm known as Maurer's algorithm for generating provable primes. ([20] p. 153)

Algorithm 3.2.5 (Maurer's algorithm)

`Prime(k)`

Input : k *a positive integer*

Output : a k*-bit prime number*

1. *If* $k \leq 20$ *then*
   (a) *Select random* k*-bit integer* n
   (b) *Use trial division for all primes less than* $\sqrt{n}$ *to determine whether* n *is prime*
   (c) *If* n *is prime then return* n
2. *Set* $c = 0.1$ *and* $m = 20$
3. *Set* $B = ck^2$
4. *If* $k > 2m$ *then select a random* s *in the interval* $[0, 1]$*, set* $r = 2^{s-1}$ *until* $(k - rk) > m$ *else set* $r = 0.5$

5. *Compute* $q = Prime([rk] + 1)$

6. *Set* $I = \frac{2^{k-1}}{2q}$

7. *Set* $success = 0$

8. *While* $success = 0$ *do*

   (a) *Select a random integer* R *in the interval* $[I+1, 2I]$ *and set* $n = 2Rq+1$

   (b) *Use trial division to determine whether* n *is divisible by any prime number* $< B$. *If not then*
   Select a random integer a *in the interval* $[2, n-2]$
   Compute $b = a^{n-2} \mod n$
   If $b = 1$ *then compute* $b = a^{2R} \mod n$ *and* $d = (b-1, n)$, *if* $d = 1$ *then* $success = 1$

9. *Return* n.

We use as implementation for this algorithm a C++ library called LIP written by Arjen K. Lenstra. From this library, we use the function PROVABLE_PRIME. This library can be found by anonymous ftp from the directory /usr/spool/ftp/pub/lenstra on flash.bellcore.com.

3.2.5 Timings in C++

In order to obtain timings in C++, we are using a library called timer.h written by Roque D. Oliviera. This simple library gives us a tool to measure time within our implementation of RSA with a precision of one millisecond. It can only be used under Unix based operating systems as it uses some of the system variables. The idea is to create an object of type timer and to initialise it at the start of the required timed portion of program and get a value in millisecond at the end of this portion of code. Here, we require a time for each exponent bit in the square and multiply algorithm. At the end of each exponentiation, the timings obtained are stored into an array which is returned from the exponentiation function.

3.3 Outline of main program

In this section, we describe briefly the general form of the main program we are using to obtain the required results for this dissertation. The algorithm is as follows.

```
FOR p,q of size p_min to p_max in steps of p_incr
   FOR e of size e_min to e_max in steps of e_incr
      Generate p and q of the required size and
      so that q is not equal to p
      Compute e
      Compute phi
      Set e=phi
      WHILE (e,phi) not equal to 1
         Generate a prime e of the required size
      END WHILE
      Compute d so that e*d is congruent to 1 modulo phi
      Save e*d mod phi into check.dat
      Save (e,phi) into check.dat
      Set message=12345678
      Encrypt message as encr
      Decrypt encr as decr
      Save data used into data.dat
      Save the timings into timing.dat
      IF decr is not equal to encr
         Save ''Error'' into check.dat
      ELSE
         Save encr and decr in decimal form
         into check.dat
      END IF
   END FOR
END FOR
```

The code for the overall implementation of the RSA algorithm and following the algorithm given above can be found in chapter A except for the LIP library which has not been included here due to its length. The results obtained using this code are analysed in chapter 5. Next, we see how the RSA algorithm can be broken by using simple techniques and more sophisticated ones.

Chapter 4

Timing Attacks and DPA

4.1 Introduction

The RSA algorithm is one of the most widely used methods when it comes to encrypting data. It is also one of the most secure encoding algorithms known to this day. It has some weaknesses though, but none has been found so far that could not be fixed. These weaknesses are not so much about the algorithm itself but the way it is implemented and used. If one is careful about how to use the algorithm, then it is a very easy and secure way to ensure information security.

There have been many attacks implemented against RSA but none has really given rise to major security breach. There are many scientists around the world who try to come up with methods to break implementations of RSA so that its security can be improved before malicious hackers even have time to attack it. RSA is widely used for very important security operations such as smartcard signatures and Internet transaction protection (eg PGP [10]) which today commonly uses a 512 bit modulus and more generally providing privacy and ensuring authenticity of digital data. It is important that the people using this algorithm spend time keeping ahead of hackers.

The first attack against RSA was the factorisation of the modulus which when known together with the public key allows us to compute the private key. When the RSA algorithm was discovered, this attack was only possible for very small moduli as the computer power was very limited and methods for factoring numbers were very inefficient. As more computing power has become available, it has become faster and faster to factor composite numbers. Also, a new method called the General Number Field Sieve was discovered by Carl Pomerance. [19] It involves breaking up the search for factors of a composite into small chunks and can therefore be performed by many computers working in parallel. It has been possible to break a 512 bits modulus in just over 7 months using 292 computers at 11 sites in 6 countries [10]. This was done using small amount of computing power and it is believed that it could be done over a period of two days using more computers dedicated solely to this task. This is a worry for people using the Internet as 512 bit the standard is used for over 95 percent of Internet commerce. There are talks about changing this standard to using a 768 bits modulus. Thus, this type of attack is not of too much concern though as it is simple to keep ahead by using bigger and bigger numbers.

Many other attacks have been developed over the years since RSA has been deployed to secure information. A brief description of some of these can be found in [16]. These, together with the factorisation of composite numbers are direct attacks on the algorithm itself.

Other more specific attacks exist depending on the way RSA is implemented. Two such attacks are known as Timing Attacks and Power Analysis. These are described later in this chapter. The timing attack is based on the software implementation of RSA and power analysis is an attack on the hardware. Another attack on the hardware takes advantage of random hardware faults. By analysing these faults, it is possible to break RSA. For example, with RSA implemented using the Chinese Remainder Theorem, a single erroneous signature exposes completely the method. This attack is described in detail in [15].

4.2 Timing attack

Timing attack has known a big breakthrough when results by Paul Kocher have were published in 1995 ([6] and [17]). The attack is known to work on different cryptosystems but we will focus here on its implications when used on RSA. This attack allows one to recover complete key information given only the running time of an operation. This concept has been known for many years but Paul Kocher's results made the complete key recovery possible.

The principle is that crytosystems take different amounts of time to work on different inputs. For computations in non-constant time, the analysis of the timing variations gives some information on the key being used and if enough is known about the implementation of the algorithm, then with the use of statistical analysis, one can recover compromising information that makes RSA insecure, hence revealing the encrypting or decrypting key.

When using RSA, we compute $R = y^{e \text{ or } d} \mod m$ depending on whether we are encoding or decoding a message y. Let us call the exponent n from now on. In this computation, m is made public and it is possible to find out what messages are being processed by an eavesdropper. In order to perform the attack, the victim must compute R above for several values of y. The attacker then knows the time taken by the computation of these y's, the messages that have been used and the modulus m. The same exponent n must be used throughout the attack so that it is feasable to work out what its value is . Also, the attacker should know the design of the target system. Here, we will assume that the algorithm used to perform the exponentiation is the square-and-multiply algorithm (see algorithm 3.2.4).

We first give a quick overview on how the attack is performed ([6]) and then we go into a more formal approach on how the attck can be implemented in practice ([5]).

In the original paper from Paul Kocher ([6]), the timing attack is carried out on the multiply step of the square and multiply algorithm. As a hypothesis, we assume that the target system has a modular multiplication algorithm that is very fast in general but can sometimes takes longer than an entire modular exponentiation. In algorithm 3.2.4, for some values of x and r, the calculation of $xr \mod m$ will be slow and because we know the implementation algorithm, it is easy to find those values for which it is slow. Due to the fact that the exponent is odd, we start our search on the second bit of this exponent, setting x_1 to 1. Now working on the second bit of the exponent, and using our samples, we can deduce that the second bit of n is zero whenever the exponentiation time is fast when $xr \mod m$ is expected to be slow and one whenever both operations are slow. This can be explained by the fact that if the exponent bit is set to one then the modular multiplication will be performed and the overall time will be increased. The operation is carried out for subsequent bits from the exponent until the full exponent has been recovered.

In [5], a more practical way of this attack that does not require such in-depth knowledge of the implementation of the algorithm is derived. We assume that the method used to carry out the multiplication runs in constant time, independently of the factors expect when the result of the multiplication is negative (see algorithm 3.2.3) and then a further addition has to be performed (this is called the reduction step). As above, we set n_1 to 1. If the second bit of the exponent is 1 then the modular multiplication step in the square and multiply algorithm will be performed together with the squaring step. There are two cases though : case one if a reduction has to be performed and case two with with no such step. If the second bit of the exponent is set to 0 though, this modular multiplication step will not be performed. We then have three samples which can be analysed using statistical analysis.

Taken from [5] is an example of a signature algorithm using RSA and a private key k,

```
M, set of messages
K, set of keys
S, set of signed messages
A : M x K -> S : (m,k) -> A(m,k), signature of m with secret key k
B={0,1}
T : M x K -> R : m -> t = T(m,k), time taken to compute A(m,k)
O : M -> B : m -> O(m), an oracle, based on what we know about the
implementation, that provides some information about
the computation of A(m,k).
```

Based on what was said previously in this section, we can devide this algorithm into two parts, namely $L(m,k)$, the computation due to the additional reduction step for the second bit of the exponent, and $R(m,k)$, the remaining computations. Then the computation times are as follows : $T(m) = T^L(m) + T^R(m)$ where $T^L(m)$ and $T^R(m)$ are the times to compute $L(m,k)$ and $R(m,k)$. The oracle O is

$$O : m \to \begin{cases} 1 & \text{if } mm^2 \text{ is done with a reduction} \\ 0 & \text{if } mm^2 \text{ is done without a reduction} \end{cases}$$

Let us define,

$$M_1 = m \in M : O(m) = 1,$$
$$M_2 = m \in M : O(m) = 0,$$
$$F_1 : M_1 \to F_1(m) = T(m),$$
$$F_2 : M_2 \to F_2(m) = T(m).$$

We have,

$$\begin{cases} F_1 = T^R & \text{if } k_2 = 0 \\ F_1 = T^R + T^L & \text{if } k_2 = 1x \\ F_2 = T^R & \forall k_2 \end{cases}$$

Having these sets, we can analyse the calculation of ϕ for F_1 and F_2 and test

$$H_0 : \phi(F_1)? = \phi(F_2)$$
$$H_1 : \phi(F_1)? \neq \phi(F_2)$$

which should give us the value of k_2. In theory, if H_0 is accepted with error porbability α then $k_2 = 1$ with error probability α. We can use the χ^2-test as our statistical test. Having this value, we carry on in the same way for the next bits until the whole key has been recovered.

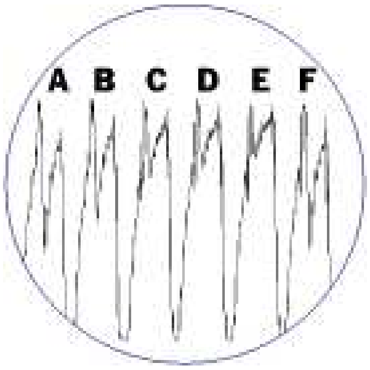
Another way of implementing the attack is by analysing the square step instead the multiplication one. This can also be seen in [5].

We next see how these timings are obtained in practice where the power consumption is being analysed and hence, noise has to be taken into account.

4.3 Power Analysis

This new attack on cryptosystems is based on the idea of timing analysis described above. As a matter of fact, the same man who discovered timing analysis, Paul Kocher, also discovered the power analysis technique. He has been working on breaking into banks and credit cards new smartcard systems. Smartcards are credit card sized devices that contain a microchip and can be used in applications such electronic cash, holding medical information, giving access to different security areas of a building, etc. This security breaking was achieved using only mid-range PCs and several thousand dollars of electronics equipment. [12] If the information held on the smartcard was to be broken into, then it would be possible to modify its content and, in the case of electronic cash for example, load counterfeit digital cash onto the card.

Smartcards internally contain a unique private key which is used to create digital signatures that are identified by the transaction machine and therefore guarantee the authenticity of the transaction being processed. It is this private key a malicous person would be interested in as if it was known then one could



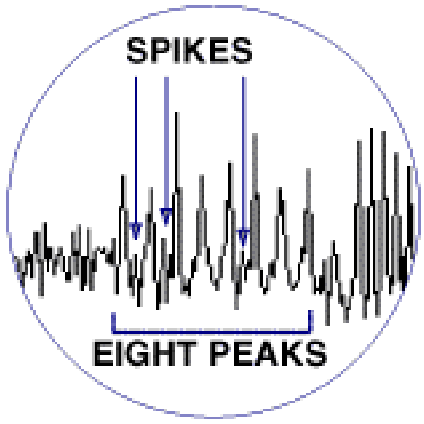
Above are six operations done by a smartcard in 1.68 μs. Operation A and F are identical, as are C and D. This series of peaks occurs whenever the card performs that series of operations. If one peak is omited at some point, then it would indicate an important change in the computation.

Figure 4.1: Electronic trace of six operations done by a smartcard [12]

create a clone of the smartcard and forge transactions that could for example transfer money from account to account. [11]

Power analysis relies on the fact that chips use electrons to do calculations and the flow of electrons can be measured using some simple electronic devices, for rough information, and more expensive devices for a more accurate reading. By watching the power consumption of the microchip on the smartcard (see figure 4.1), it is possible identify what operations are being performed during the process of digital signature exchange, obtain some timings on the duration of those operations and hence to break the secret key using timing analysis for example. In earlier versions of smartcard, it has been possible to extract this key by simply watching an oscilloscope graphing the power consumption of a card and identifying the peaks and valleys on this graph as each operation has a distinct patterns. [11] This is known as simple power analysis (SPA). It can be used, for example, to break RSA implementations by showing differences between modular multiplications and exponentiations. This relates to timing attacks described above.

However, new smartcards used today have got some more sophisticated microchips which do not allows one to identify such patterns by a simple reading (see figure 4.2). Some analysis has to be done in order to obtain the information required. This type of analysis is known as differential power analysis (DPA).



The sequence of eight peaks indicates a part of an encryption operation that protects information on the card. The presence or absence of spikes between these peaks gives analysts a piece of the encryption key, of which further, similar analysis may reveal additional pieces.

Figure 4.2: Electronic trace of part of an encryption operation [12]

[7] DPA is a lot more powerful than SPA for it uses statistical techniques and error correction techniques to get information about the secret key. There are two phases to DPA, namely information collection and data analysis. The collection of data can be done using SPA by getting the device's power consumption. This data collection has to be done for several thousand messages. Then statistical analysis of the data collected is necessary in order to find the correlation between the power consumption and the value of the secret and in the process eliminate the noise on the power trace.

Hence, in the example of RSA and relating the attack to timing analysis, the analysis of the power trace gives us information on what operations are being performed in the RSA implementation and by analysing the timing once the noise has been removed through statistical analysis and obtained a clearer trace, we can identify when the encryption starts and find the value of the secret key. This attack has been implemented by Paul Kocher and more information can be found in [8].

4.4 Possible Countermeasures

4.4.1 Timing attack

For the RSA algorithm, a technique called blinding can be used to fight against timing attacks. The approach was suggested by Ron Rivest, one of the inventors of RSA. Let us assume we want to calculate $y \equiv x^d \mod n$ where x is the message, d is the secret key, n is the modulus and y is the result. This is then done as follows,

1. Generate a secret random number r between 0 and $n-1$.
2. Compute $x' \equiv xr^e \mod n$, where e is the public exponent.
3. Compute $y' \equiv (x')^d \mod n$ with the ordinary RSA implementation.
4. Compute $y \equiv y'r^{-1} \mod n$.

We see that the result is as expected by noting that $r^{ed} \equiv r \mod n$. Since step 3 involves a random secret x', its running time cannot be correlated to the input x.

Another way to protect our implementation against timing attack is to make sure the reduction step in the Head's algorithm is always performed even if we do not use it afterwards. However, it is not guaranted that the systems will be protected against any type of timing attack, only those which use the reduction of the multiplication algorithm. [5]

A recent improvement was proposed by J.F. Dhem in [18] and consists in chaining together several modular multiplications with only one extra reduction being performed after the last multiplication. It is very interesting as it suppresses the attacks main target.

4.4.2 DPA

DPA can only be used against smartcards and small devices. It is no real threat to big systems like PCs at the time as the noise created in this case would be far too important to allow statistical analysis to find any correlation between the power consumption and the secret key.

There are a few possible solutions to avoid a DPA attack on smartcards and similar systems. Basically, what one would like to achieve is masking the power consumed by doing the encryption.

This could be done by adding a secondary circuit on the chip that would do some calculations on random numbers. We are not sure though whether it would be possible to give enough randomess to resist today's most advanced statistical attacks. Indeed, random calculations come to an average over the time and could probably be removed by DPA. [11]

Another solution would be to add a parallel circuit that would act as a mirror on the encryption calculations. This means that when one circuit is doing the real encryption process on 1101 then the other one would do the same operation on 0010 which, in theory, should smooth out the power consumption as the combination of the two circuit's power use should be more constant. Again, it is unclear whether it would be enough to defeat DPA. [11]

Another way of making encryption DPA proof would be to modify the software used to implement the encryption process by assuming that information will effectively be available to a potential hacker. Some approaches have been developped in order to achieve this and in the case of high hardware leakage levels, leak reduction and some of the masking techniques described above can be used. [7]

Chapter 5

Timing analysis

The code shown in appendix A is an implementation of the RSA algorithm. It has been designed in order to obtain time intervals within the modular exponentiation of the encrypting and decrypting phases of RSA. The algorithm used to implement the modular exponentiation is called square-and-multiply (algorithm 3.2.4). We have obtained information on how long the exponentiation takes for each bit in the exponent. Each step in the square-and-multiply algorithm corresponds to one bit in the exponent being processed. From chapter 4, we know that these time intervals can give some or all the information necessary to find the exponent.

First of all, we show the format of the data we have recorded and then we analyse it and obtain results on the correlation between the time taken in one step of square-and-multiply and the bit in the exponent being computed.

5.1 The data

The main data obtained is the time intervals within the modular exponentiation function. As well as those timings, we have obtained the time taken from the calculation of n, $\phi(n)$, e and d.

The data is stored in three files :

1. data.dat which contains the data used when running RSA namely the size of p and q, p and q, n (the modulus), $\phi(n)$, size of e, e and d (the encrypting and decrypting keys).

2. check.dat contains the main checks needed within RSA such as $(e, \phi(n)) = 1$, $1 \equiv ed \mod n$, the original message, the ecnrypted message and the

Size of p in bits	50
p	10110000011100010001101001000001011101110000111111
Size of q in bits	50
q	10101001011010000111111010000000110110101000 10101
n	11101001100001010101111111111000110010111001110101100010101 ... 1010110011100001011100101001110000101011
$\phi(n)$	11101001100001010101111111111000110010111001110010110101110 ... 0100011001111011011010110000101011011000
Size of e in bits	20
e	10110010110000110001
d	110110111110001000001111101110000111011100010111111101110011 ... 0000100101100101000101010001000000111001

Table 5.1: First entry in the file data.dat

$(e, \phi(n))$	1
$ed \mod n$	1
Message	12345678
Encrypted message	111111111111010000101011101010010010110101111110001101000 0100 ... 111110001101011001010010100 1001
Decrypted message	12345678

Table 5.2: First entry in the file check.dat

decrypted message. If the decrypted message does not equal the original message then the message 'Error' is output instead of the decrypted message.

3. timing.dat contains the time intervals (in seconds) taken at the different stages of the RSA algorithm.

Details of the format of the three files are shown in tables 5.1, 5.2 and 5.3.

We have obtained results for p and q of length varying from 50 to 290 bits in steps of 5 bits and e of length varying from 20 to 100 bits also in steps of 5 bits. The code was run on a computer with two 150Mhz SPARC processors running Solaris 2.7 and 320 Mb RAM. We will now look at a possible correlation between the time intervals obtained and the public key.

Time taken to compute n in seconds	0
Time taken to compute $\phi(n)$ in seconds	0
Time taken to compute e in seconds	0
Time taken to compute d in seconds	0.02
a)	Encr
b)	0.04 0.03 0.01 0.03 0.04 0.05 0.01 0.02 0.02 0.03 0.04 0.05 0.03 0.04 0.03 0.02 0.05 0.04 0.03 0.04
c)	Decr
d)	0.04 0.03 0.02 0.05 0.05 0.04 0.03 0.02 0.03 0.02 0.03 0.05 ... 0.02 0.02 0.03 0.05 0.02 0.05 0.02 0.05 0.02 0.03 0.02 0.05 ... 0.02 0.04 0.03 0.02 0.05 0.05 0.02 0.04 0.02 0.03 0.04 0.03 ... 0.02 0.03 0.02 0.05 0.05 0.02 0.03 0.04 0.05 0.04 0.03 0.05 ... 0.04 0.05 0.05 0.05 0.04 0.05 0.02 0.05 0.03 0.02 0.02 0.05 ... 0.05 0.05 0.02 0.05 0.05 0.05 0.02 0.02 0.03 0.02 0.05 0.05 ... 0.04 0.03 0.05 0.05 0.04 0.05 0.05 0.02 0.02 0.03 0.02 0.03 ... 0.05 0.02 0.02 0.03 0.05 0.04 0.05 0.05 0.04 0.03 0.05 0.04 ... 0.03 0.04 0.05

a) Start of the timings for the encrypting process
b) Timings for each bit in the exponent in the square-and-multiply algorithm
c) Start of the timings for the decrypting process
d) Timings for each bit in the exponent in the square-and-multiply algorithm

Table 5.3: First entry in the file timing.dat

5.2 Analysing the data

From the section on timing attacks in chapter 4, we know that the square-and-multiply algorithm can be attacked by looking at the time taken to compute one step of the modular exponentiation, given that each step corresponds to one bit in the exponent being processed. Indeed, we have seen that the modular multiplication in the algorithm is only computed if we are processing a one and that for a zero, we jump directly to the modular squaring step of the algorithm. This creates time differences that are correlated to the value of the bit in the exponent being used. In our code, since we have placed timing tags at the start and finish of each step of square-and-multiply, we can expect to see similar results and we hope to be able to show that we can then recover the full value of the exponent.

We notice that the time intervals we have obtained are of the order of the millisecond. Although this is not a very fine unit, we can still see that there are some variations in time during the encryption and decryption processes. In order to see more clearly the data, we have rearranged it in the form of graphs as shown

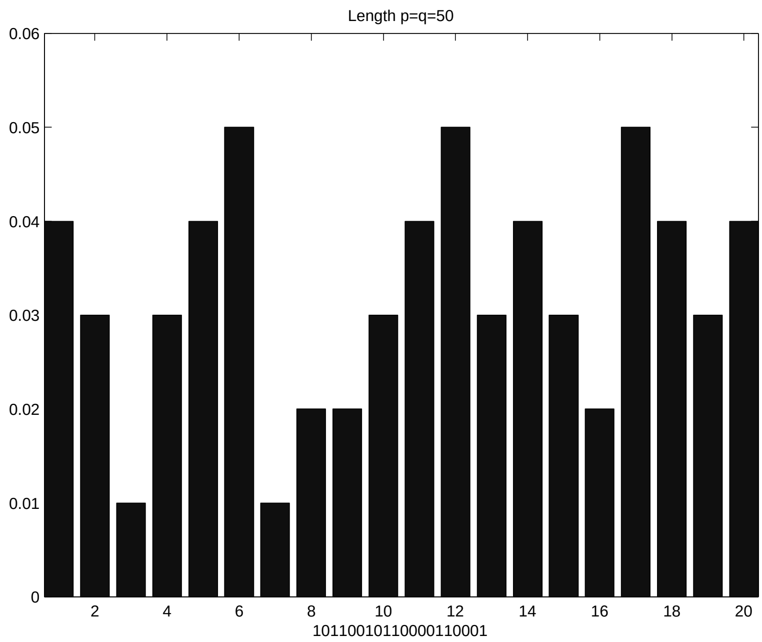


Figure 5.1: Timings for an encrypting key of size 20 bits with p and q of size 50 bits

in figure 5.1. We have created graphs for the whole range of lengths of p and q and for e of length 20 and 60 bits in order to compare :

1. The variations in the timings as the size of p and q increases.

2. The variations in the timings as the size of e increases.

3. The correlation between the time intervals and the value of the key.

These graphs can be found in appendix B.

On graph 5.1, we see represented on the x-axis the bits of the exponent being processed, 1 corresponding to the least significant bit and 20 to the most significant bit. The value of the exponent is shown below the x-axis. On the y-axis, we have represented the time taken to process one bit of the exponent in the square-and-multiply modular exponentiation algorithm. We now look for some correlation between the bit being processed in the exponent and the time taken to perform this operation.

Looking at figure 5.2, for p and q of length 65 bits, we see that there are solely values less than 0.03 seconds or values grater than 0.06 seconds. Now looking at the exponent, going from the least significant bit to the most significant bit,

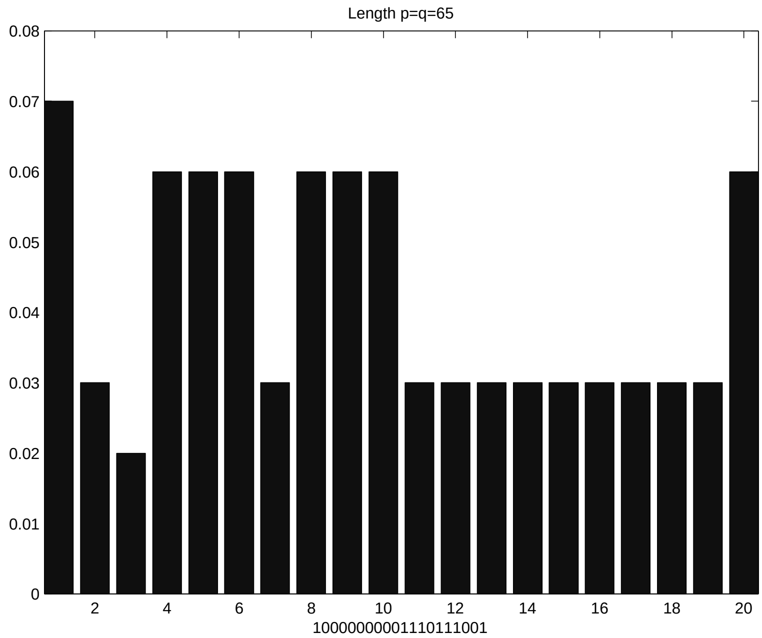


Figure 5.2: Timings for an encrypting key of size 20 bits with p and q of size 65 bits

we see that those values below 0.03 seconds correspond to the value of the bit in the exponent equal to zero and those values above 0.06 seconds correspond to the value of the bit in the exponent equal to one. Looking at other graphs, we see that a similar conclusion can be drawn, or in other words, that it is possbile to differentiate between time intervals corresponding to a one and those corresponding to a zero. Namely, for smaller times, we are computing a bit with value zero and for bigger ones, a bit with value one.

However, for small values of p and q, it is difficult to tell where the limit occurs as to when we conclude a time interval corresponds to a one or a zero. By averaging the time intervals representing zeros and the time intervals representing ones as shown in figures B.23, B.24 and B.25, we obtain some useful information. We first note that the time taken to process a zero stays close to the average value calculated. For example, for p and q of length 50 bits, the average found is 0.02 seconds for a zero and, by looking at figure 5.1, we see that the maximum value for a zero is 0.03 seconds. Looking at other graphs for other values of p and q, we can generalise this result. Hence, it is possible to find what times correspond to a zero in the exponent by looking at the average value, which could have been calculated prior to the attack, and then deduce what time intervals correspond to a one in the exponent.

We note that some values that we would interpret as a zero being processed in fact represent a one. This is due to the fact that we have run the code on a shared machine and this may correspond to the processor not having a load as important at the time of computation, which would have speeded up the calculation of one step in the square-and-multiply algorithm. Such an example can be found in figure B.1 for p and q of length 55. Indeed, the third bit from the right has a time interval we would expect to correspond to a zero but in fact is a one.

Comparing at the average obtained for the encrypting key of length 20 and 60 bits in figures B.23 and B.24 for p and q of length 50 to 105 bits, we note that there are no differences in the time taken to compute one step of square-and-multiply. The overall time for exponentiation becomes bigger though, as there are more bits being processed. This result was expected as there is no difference in the processing of exponents of different lengths but the number of steps taken in the modular exponentiations.

We see that as p and q increase, the gap between the time intervals for a zero and a one increases significantly. For example, looking at figure 5.3, corresponding to p and q of length 195 bits, we see very clearly that for a time below 0.2 seconds, a zero is being processed and for a time above 0.35 seconds, a one is being processed, with a 0.15 seconds difference between the two time intervals. Looking at the averages obtained in figure B.25, we see that it is indeed the case that the gap increases proportionally to the length of p and q. This increase in the gap makes it easier to obtain information about the value of the bit in the exponent being processed as we can then read it straight from the graph.

5.3 Conclusion

We have seen above that it is possible to read what the value of the exponent is simply by looking at the time intervals obtained from the modular exponentiation algorithm. For smaller values of p annd q, it is more difficult to differentiate between zeros and ones but as the length of p and q increases, the gap in the time interval representing a one and a zero increases significantly making it very easy to obtain the value of the exponent. For smaller values of p and q, we can obtain the average time interval for a zero and, by noting that values for a zero being processed stay close to this average, obtain the bits in the exponent that

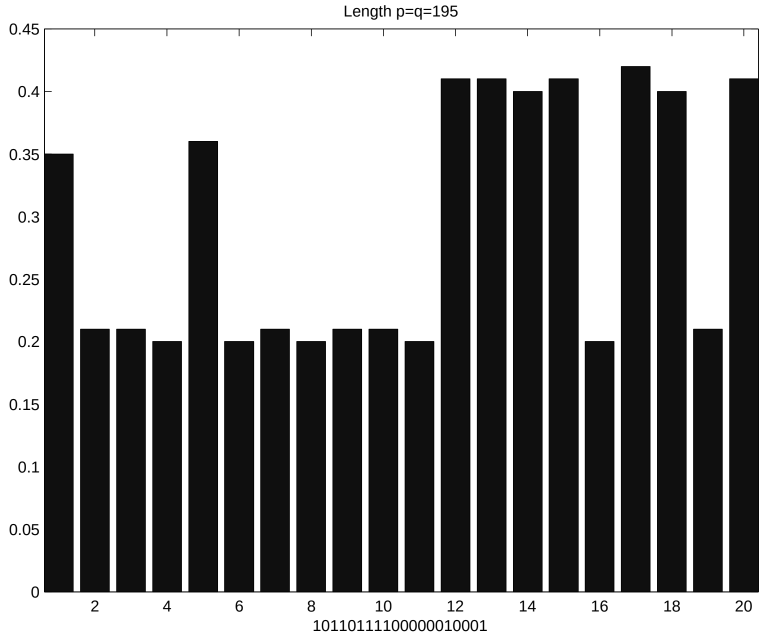


Figure 5.3: Timings for an encrypting key of size 20 bits with p and q of size 195 bits

are zeros and then deduce which are ones. These averages could be computed prior to the attack.

The length of the key only has an influence on the overall time of execution of the modular exponentiation algorithm but the time intervals for computing one step of the square-and-multiply algorithm are the same regardless of the length of the exponent. Hence, making the encrypting (or decrypting) key bigger only increases the overall time taken to find its value but does not affect the complexity of the task.

We have seen in one occasion that one of the timing intervals obtained was not what we were expecting. Indeed, the time taken to process a one was of the order of the average time taken to process a zero in a case where p and q were relatively small. This was due to the fact that the computer on which the calculations were done was shared between many users and hence, the processor workload varied significantly with time. If we were to run the code on a processor solely designated to our implementation of RSA, it is fair to say that such errors would not occur.

In our analysis, we have used the data obtained for the public key. This was only done because we knew what its length was as it was recorded throughout the execution of the code. Similar results can be obtained by using the results obtained for the secret key.

The time intervals we have obtained imply that we know of the implementation of the modular exponentiation. Here, as opposed to the timing attack described in chapter 4, we do not have to use statistical analysis. The results we have show that indeed there is a basis for timing attack and DPA as time intervals within the modular exponentiation are directly related to the value of the exponent.

The traces that one would obtain using DPA are in fact a noisy version of those time intervals we have obtained using the code presented in appendix A. By the use of statistical analysis, it would be possible to remove this noise and hence obtain data that would be clear enough to corelate it to the exponent in the same way we have done above. However, we do not know how noisy the traces we obtain could be in order to be able retreive enough information to recover the secret key.

Chapter 6

Conclusion and further work

6.1 Conclusion

The RSA algorithm, although a widely used algorithm and a very secure one has some weaknesses. It is very important for people who are implementing RSA to know what these are and for this reason, there are scientists all around the world who are trying to find different ways of breaking the algorithm. The biggest problem known to date with the algorithm is the factorisation of the modulus used in the encryption and decryption processes. This is one that evolves with the growth of computing power available as new techniques have been found to factor numbers using computers working in parallel. This is easily dealt with though as it is very possible to use bigger and bigger integers as we follow the evolution of computing power.

More problematic attacks are those on the design and implementation of RSA. One such attack, known as timing analysis, analyses the time taken to perform the modular exponentiation for each bit in the private key. Indeed, when performing the modular exponentiation, the time taken to compute a one in the exponent is bigger than the time taken to compute a zero due to an extra modular multiplication in the square-and-multiply algorithm. By knowing what messages are being encrypted (through the use of an eavesdropper for example) and also by having the value of the modulus (available with the public key), it is possible to recover the secret key by the use of statistical analysis. However, this involves having some insight on the way the modular exponentiation is being performed. A method has been developped on the same principle using information on whether a modular reduction is being performed or not. This method works even when the implementation of the modular exponentiation is not known. [5]

Another attack, known as differential power analysis, is performed on small devices such as smardcards. This attack consists in analysing the power consumption of the microchip on the card. From the power probe and with the use of statistical analysis, it is possible to identify what operations are being performed and also obtain the time taken bewteen each operation. This clearly relates to timing analysis as once the time intervals have been obtained, we can implement the timing attack described in chapter 4. The differential power analysis method has been implemented and tested on several smartcards by Paul Kocher using equipment worth a few hundred dollars.

Some coutermeasures have been developped in order to fight these new attacks on the hardware or software of implementations of the RSA algorithm but none have be proven to be very efficient so far, especially as far as differential power analysis is concerned.

In order to show the correlation between the time intervals in the square-and-multiply algorithm and the exponent, we have implemented RSA in C++. We have obtained the time taken in the modular exponentiation for each bit in the public and private key for a fixed plaintext message, varying the size of p and q from 50 to 290 bits and the size of the encrypting key from 20 to 100 bits. Using these results, we have shown that there is a correlation between those time intervals and the bits being processed in the public key. As the size of p and q increases, the timing traces reveal clearly the sequence of binary digits in the exponent. For smaller values of p and q though, we need to do more work by obtaining the average time taken to process a zero as we have noticed that in this case, the time intervals stay very close to this average value. We then find the bits in the exponent that are zero and deduce where the ones are. Also, we have seen that the length of the exponent we are using does not affect the complexity of the analysis of the data but only the time it takes to obtain the exponent.

We can then conclude that there is a basis for timing attack and differential power analysis as the time intervals are directly linked to the value of the exponent. The results we have obtained are a clear, noiseless version of what one would get when analysing the power consumption of a device implementing RSA. Hence, by looking at the power consumption and identifying the different operations performed and time intervals by the use of statistical analysis, we could expect to be able to recover the secret key.

6.2 Further work

We have seen that it is possible to recover the secret key by using DPA. Since we have obtained the time intervals through the use of our code and we know these are directly linked to the exponent, it would be interesting to add some noise to these time intervals and then try to recover the original data from this noisy data. We could then see how much noise could be added before it is impossible for us to obtain any useful information on the exponent. This would allow us to tell what type of devices are at risk. According to Paul Kocher, computers such as PCs could not in practice be broken into as too nuch noise is present on the power probe and only small devices are affected.

Having obtained data on only one processor, it would be interesting to run the code on different processors and see how the time intervals vary from one to the other. In particular, we would like to know to have a processor solely designated to the running of our code without any other operations being performed on it or processes running in the background or shared with other users.

Also, it would be interesting to run the code on different processors and look at the power consumption and see if we can identify the different operations being performed. This way, we could identify the different probes we would expect from different processors. We could then relate these results to those obtained theoretically from applying noise to the data we have obtained for this dissertation. We could also use only part of the processing power available and gradually use more and more of it to see how it affects the power consumed.

From this possible analysis, we may be able to draw some new, efficient countermeasures against DPA. We have seen that the existing ones have not been proven to be very secure so there is certainly room for more inverstigation in this domain as it is a real threat to the RSA algorithm as implemented on smartcards.

Bibliography

[1] Kenneth H. Rosen
Elementary Number Theory and its Applications
Addison-Wesley, 1992.

[2] Albrecht Beutelspacher
Cryptology
The Mathematical Association of America, 1994.

[3] Neal Koblitz
A Course in Number Theory and Cryptography
Springer, 1994.

[4] Peter Giblin
Primes and Programming : An Introduction to Number Theory with Computing
Cambridge University Press, 1993.

[5] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater, J.-L. Willems
A practical implementation of the timing attack
UCL Crypto Group Technical Report Series, June 15, 1998
http://www.dice.ucl.ac.be/crypto/

[6] Paul C. Kocher
Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems
Crypto' 96, volume 1109 of Lecture Notes in Computer Science, pp. 104-113, Springer-Verlag, 1996
http://www.cryptography.com/

[7] Paul Kocher, Joshua Jaffe, Benjamin Jun
Introduction to Differential Power Analysis and Related Attacks
http://www.cryptography.com/

[8] Paul Kocher, Joshua Jaffe, Benjamin Jun
Differential Power Analysis
Advances in Cryptology : Proceedings of CRYPTO '99, Springer-Verlag, August 1999

[9] Ivars Peterson
Power Cracking of Cash Card Codes
Science News Online, June 20, 1998
http://www.sciencenews.org/

[10] Sara Robinson
Researchers Demonstrate Computer Code Can Be Broken
The New York Times, August 27, 1999
http://www.nytimes.com/

[11] Peter Wayner
Cryptographers Discuss Finding Of Security Flaw in 'Smart Cards'
The New York Times, June 10, 1998
http://www.nytimes.com/

[12] Peter Wayner
Code Breaker Cracks Smart Cards' Digital Safe
The New York Times, June 22, 1998
http://www.nytimes.com/

[13] John Markoff
U.S. Data-Scrambling Code Cracked With Homemade Equipment
The New York Times, June 17, 1998
http://www.nytimes.com/

[14] Ivars Peterson
Exploiting faults to break smart-card cryptosystems
Science News Online, February 1, 1997
http://www.sciencenews.org/

[15] Dan Boneh, Richard A. DeMillo, Richard J. Lipton
On the Importance of Eliminating Errors in Cryptographic Computations
http://www.cs.standford.edu/

[16] Dan Boneh
Twenty Years of Attacks on the RSA Cryptosystem
http://www.cs.standford.edu/

[17] John Markoff
Secure digital transactions just got a little less secure
The New York Times, December 11, 1995

[18] J.F. Dhem
Design of an efficient public-key cryptography library for RISC-based smart-cards
PhD thesis, Universite Catholique de Louvain
UCL Crypto Group, May 1998

[19] Carl Pomerance
Al Tale of Two Sieves
Notices of the American Mathematical Society, December 1996

[20] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone
Handbook of Applied Cryptography
CRC Press, October 1996

Appendix A

Code

Here is a listing of the code that I have written or some code modified or copied from the internet. The original code for the class largenumber can be found at www.catachan.demon.co.uk/Programming/Large/.

A.1 prog.cc

```
/************************************************************/
/*    Author : Brice Canvel                                 */
/************************************************************/
/*    Purpose : Use of the classes LargeNumber              */
/*              timer and lip in order to obtain timings    */
/*              involved in the use of the RSA              */
/*              algorithm                                   */
/************************************************************/

#include <fstream.h>
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "largenumber.h"
#include "lip.h"
#include "timer.h"

/***********************************************************/
/*    V A R I A B L E S                                    */
/***********************************************************/

// p and q : two prime numbers used in the RSA algorithm
// n = p*q
// phi = (p-1)*(q-1)
// msg : Message to encrypt of type LargeNumber (in binary form)
// encr : Encrypted message
// decr : Decrcyppted message
// e : Encrypting key
// d : Decrypting key

LargeNumber p,q,phi,encr,decr,msg,e,n,d;

// np, nq : Size of p and q (in bits)
// ne : Size of e (in bits)

long np,nq,ne;

// Variables used in recording the time taken for certain computations

Timer n_time,phi_time,d_time,e_time;
double *encr_time,*decr_time;

// zn : Variable of type verylong used to store random prime numbers found
//      using the function PROVABLE_PRIME

verylong zn ;

// data : Relates to the files data.dat containing the data used during
//        encryption and decryption processes
// timing : Relates to the file timing.dat containing the times recorded
//          during encryption and decryption processes
// check : Relates to the files check.dat containing the checks done during
//         encryption and decryption processes

fstream data,timing,check;

// Various counters used within the main program

int p_counter,e_counter,p_min,p_max,p_incr,e_min,e_max,e_incr;

/************************************************************/
/*    P R O V A B L E _ P R I M E                           */
/************************************************************/
```

```
/*
   void PROVABLE_PRIME ( long k, verylong *zn ) :

   Finds a random prime number of size k bits using Maurer's algorithm.
   This function uses the class lip in which is defined the type
   verylong, a decimal representation of very long integers.

   */

void PROVABLE_PRIME ( long k, verylong *zn )
{
  double c, r, s;
  int success;
  long B, m, n, p, sqrtn;
  verylong zI = 0, zR = 0, za = 0, zb = 0, zc = 0;
  verylong zd = 0, zk = 0, zl = 0, zq = 0, zu = 0;

  srand(time(NULL));
  zrstarts(time(NULL));
  if (k <= 20) {
    do {
      n = OddRandom(k);
      sqrtn = sqrt(n);
      zpstart2();
      do p = zpnext(); while (n % p != 0 && p < sqrtn);
    } while (p < sqrtn);
    zintoz(n, zn);
  }
  else {
    c = 0.1;
    m = 20;
    B = c * k * k;
    if (k > 2 * m)
      do {
        s = rand() / (double) RAND_MAX;
        r = pow(2.0, s - 1.0);
      } while (k - r * k <= m);
    else
      r = 0.5;
    PROVABLE_PRIME(r * k + 1, &zq);
    zone(&za);
    zlshift(za, k - 1, &zk);
    zcopy(zq, &za);
    zlshift(za, 1l, &zl);
    zdiv(zk, zl, &zI, &za);
    zsadd(zI, 1l, &zl);
    zlshift(zI, 1l, &zu);
    success = 0;
    while (!success) {
      do zrandomb(zu, &zR); while (zcompare(zR, zl) < 0);
      zmul(zR, zq, &za);
      zlshift(za, 1l, &zb);
      zsadd(zb, 1l, zn);
      zcopy(zR, &za);
      zlshift(za, 1l, &zR);
      zpstart2();
      p = zpnext();
      while (zsmod(*zn, p) != 0 && p < B) p = zpnext();
      if (p >= B) {
        zcopy(*zn, &zc);
        zsadd(zc, - 2l, &zb);
        do
          zrandomb(*zn, &za);
        while (zscompare(za, 2l) < 0 || zcompare(za, zb) > 0);
        zsadd(*zn, - 1l, &zc);
```

```
        zexpmod(za, zc, *zn, &zb);
        if (zscompare(zb, 1l) == 0) {
          zexpmod(za, zR, *zn, &zb);
          zcopy(zb, &zd);
          zsadd(zd, - 1l, &zb);
          zgcd(zb, *zn, &zd);
          success = zscompare(zd, 1l) == 0;
        }
      }
    }
  }
  zfree(&zI);
  zfree(&zR);
  zfree(&za);
  zfree(&zb);
  zfree(&zc);
  zfree(&zd);
  zfree(&zk);
  zfree(&zl);
  zfree(&zq);
  zfree(&zu);
}

/**********************************************************/
/*    D E C 2 B I N                                       */
/**********************************************************/

/*
  LargeNumber dec2bin ( verylong var ) :

  Converts a decimal number of type verylong into a binary number
  of type LargeNumber.
  This function uses both classes lip and LargeNumber.

  */

LargeNumber dec2bin ( verylong var )
{
  LargeNumber znl;
  verylong zn=var;
  int i=1;

  while ( ziszero(zn) == 0 )
    {
      znl.expand(i);
      if ( z2mod(zn) == 1 )
znl.number[i-1]=1;
      else
znl.number[i-1]=0;
      zrshift(zn,1l,&zn);
      i++;
    }
  znl.contract();

  return znl;
}

/**********************************************************/
/*   M A I N                                              */
/**********************************************************/

int main()
{

  // Opens all files used within the program
  data.open("data.dat",ios::app);
```

```
  timing.open("timing.dat",ios::app);
  check.open("check.dat",ios::app);

  p_min=50; // Minimum size of p in bits
  p_max=300; // Maximum size of p in bits
  p_incr=5; // Increment value of the size of p in bits

  e_min=20; // Minimum size of e in bits
  e_max=100; // Maximum size of e in bits
  e_incr=5; // Increment value of e in bits

  // Loop for the size of p (in bits) going from p_min to p_max in steps of p_incr
  for (p_counter=p_min;p_counter<=p_max;p_counter=p_counter+p_incr)
    {

      np=p_counter; // Defines the size of p in bits
      nq=p_counter; // Defines the size of q in bits (equal the size of p)

      // Loop for the size of e (in bits) going from  e_min to e_max in steps of e_incr
      for (e_counter=e_min;e_counter<=e_max;e_counter=e_counter+e_incr)
{
  ne=e_counter;

  // Generates random prime p of size np bits
  PROVABLE_PRIME(np,&zn); /* Uses function PROVABLE_PRIME which works
     with numbers of type verylong
     (defined in the class lip)
     */
  p=dec2bin(zn); // Converts number found from type verylong to type LargeNumber
  q=p;

  // Generates a random prime q of size nq in bits (not equal to p)
  while (q == p)
    {
      PROVABLE_PRIME(nq,&zn);
      q=dec2bin(zn);
    }

  // Computes n=p*q and records the time taken
  n_time.start();
  n_time.start();
  n=p*q;
  n_time.stop();

  // Computes phi=(p-1)*(q-1) and records the time taken
  phi_time.start();
  phi=(p-1)*(q-1);
  phi_time.stop();

  // Computes a value for e so that the greatest common divisor
  // of e and phi equals 1
  e=phi;
  while (gcd(e,phi) != 1)
    {
      e_time.start();
      PROVABLE_PRIME(ne,&zn);
      e=dec2bin(zn);
      e_time.stop();
    }

  // Computes the value of d (inverse of e modulo phi) and records the time taken
  d_time.start();
  d=modinv(e,phi);
  d_time.stop();

  // Outputs to the file check.dat the value e to the power d modulo phi
```

```
  // which should equal 1 and also the gratest common divisor
  // of e and phi which should also equal 1 (This is for checking purposes)
  check<<MultiplyModN(e,d,phi,Sqrt(phi))<<" ";
  check<<gcd(e,phi)<<" ";

  // Sets the message
  msg=12345678;

  // Encrypts the message and records the time and encrypted message
  encr_time=new double [e.length()];
  encr=PowerRule(msg,e,n,encr_time);

  // Decrypts encrypted message and records the time and decrypted message
  decr_time=new double [d.length()];
  decr=PowerRule(encr,d,n,decr_time);

  // Outputs to the file data.dat the data used for the encryption and decryption processes
  data<<np<<" "<<p<<" "<<nq<<" "<<q<<" "<<n<<" "<<phi<<" "<<ne<<" "<<e<<" "<<d<<endl;

  // Outputs to the file timing.dat the timings obtained for the encryption and decryption processes
  timing<<n_time.cpuTime()<<" "<<phi_time.cpuTime()<<" "<<e_time.cpuTime()<<" "<<d_time.cpuTime()<<" Encr ";

  for (int k=0;k<e.length();k++)
    timing<<*(encr_time+k)<<" ";
  timing<<"Decr ";

  for (k=0;k<d.length();k++)
    timing<<*(decr_time+k)<<" ";
  timing<<endl;

  // Outputs to the file check.dat the checks done during the encryption and decryption processes
  if (decr != msg)
    check<<"Error"<<endl;
  else
    check<<msg.to_int()<<" "<<encr<<" "<<decr.to_int()<<endl;

  delete encr_time;
  delete decr_time;

}
    }

  // Closes all files
  data.close();
  timing.close();
  check.close();
}
```

A.2 largenumber.h

```
// Arbitrarily large numbers
//
// A LargeNumber object should from any external point store its unsigned
// number in positional code in the range [0..sig] of its number array.
// This code should always lack leading 0s except when representing zero itself.
// If the represented number is negative then the negative flag should be set
// otherwise it should not be set.  The max should be equal to one less than
// the number array size and sig should never be greater than max.
//
// Author  Matthew Caryl
// Created 13.3.97

#include <iostream.h>

class LargeNumber
{
    public:
    LargeNumber(void);
    LargeNumber(long n);
    LargeNumber(const LargeNumber& n);
    ~LargeNumber(void);

    char to_char(void) const;
    short to_short(void) const;
    int to_int(void) const;
    long to_long(void) const;

    int even(void) const;
    int odd(void) const;
    int length(void) const; // in bits
    int bit(unsigned int p) const; // p'th bit (from zero)
    int zero() const; // is zero
    int sign(void) const; // is negative

    void truncate(int n); // reduce to lower n bits
    void complement(void); // * -1

    int operator==(const LargeNumber& n) const;
    int operator!=(const LargeNumber& n) const;
    int operator<(const LargeNumber& n) const;
    int operator<=(const LargeNumber& n) const;
    int operator>(const LargeNumber& n) const;
    int operator>=(const LargeNumber& n) const;

    LargeNumber& operator=(const LargeNumber& n);
    LargeNumber& operator+=(const LargeNumber& n);
    LargeNumber& operator-=(const LargeNumber& n);
    LargeNumber& operator<<=(int n);
    LargeNumber& operator>>=(int n);
    LargeNumber& operator++(void);
    LargeNumber& operator--(void);
    LargeNumber  operator++(int);
    LargeNumber  operator--(int);
    LargeNumber& operator*=(const LargeNumber& n);
    LargeNumber& operator/=(const LargeNumber& n);
    LargeNumber& operator%=(const LargeNumber& n);
    // no change of sign for following operators
    LargeNumber& operator&=(const LargeNumber& n);
    LargeNumber& operator|=(const LargeNumber& n);
    LargeNumber& operator^=(const LargeNumber& n);

    LargeNumber operator+(const LargeNumber& n) const;
    LargeNumber operator-(const LargeNumber& n) const;
    LargeNumber operator*(const LargeNumber& n) const;
```

```
    LargeNumber operator/(const LargeNumber& n) const;
    LargeNumber operator%(const LargeNumber& n) const;
    // no change of sign for following operators
    LargeNumber operator&(const LargeNumber& n) const;
    LargeNumber operator|(const LargeNumber& n) const;
    LargeNumber operator^(const LargeNumber& n) const;
    LargeNumber operator<<(int n) const;
    LargeNumber operator>>(int n) const;

    friend ostream& operator<<(ostream& s, const LargeNumber& n);
    friend istream& operator>>(istream& s, LargeNumber& n);

    char* number;
    int negative;
    int sig;
    int max;

    // unsigned operators
    int smaller(const LargeNumber& n) const; // unsigned
    int greater(const LargeNumber& n) const; // unsigned
    void expand(int n); // ensure n'th bit exits
    void contract(); // remove leading 0s
    void plus(const LargeNumber& n); // unsigned
    void minus(const LargeNumber& n); // unsigned
};

LargeNumber gcd( const LargeNumber &X, const LargeNumber &Y );
LargeNumber modinv( const LargeNumber &a, const LargeNumber &m );
LargeNumber MultiplyModN (LargeNumber x,LargeNumber y,LargeNumber m,LargeNumber sqrt_m);
LargeNumber PowerRule(LargeNumber x,LargeNumber n,LargeNumber m);
LargeNumber Sqrt(LargeNumber n);
LargeNumber power(LargeNumber x,LargeNumber n);
LargeNumber is_prime(LargeNumber testNum);
long OddRandom(long bit_length);
```

A.3 largenumber.cc

```
// Arbitrarily large numbers
//
// Author  Matthew Caryl
// Created 13.3.97

#include <iostream.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "largenumber.h"
#include "timer.h"

const int false=(1 == 0);
const int true=(1 == 1);

int maximum(int a, int b);
int minimum(int a, int b);
long pow(long a, long b);

const unsigned int BIT_INCREMENT = 128;

int maximum(int a, int b)
{
    return a > b ? a : b;
}

int minimum(int a, int b)
{
    return a < b ? a : b;
}

long pow(long a, long b)
{
    long a1 = a;
    long b1 = b;
    long c = 1;

    while (b1 >= 1)
        {
        while (b & 1 == 0)
            {
            b1 = b1 / 2;
            a1 = a1 * a1;
            }
        b1 = b1 - 1;
        c = c * a1;
        }
    return c;
}

void LargeNumber::contract()
{
    while (number[sig] == 0 && sig > 0)
        sig--;
}

LargeNumber::LargeNumber(void)
{
    number = new char[BIT_INCREMENT];
    number[0] = 0;
    negative = false;
    max = BIT_INCREMENT - 1;
    sig = 0;
```

```
}

LargeNumber::LargeNumber(long n)
{
    negative = n < 0 ? true : false;
    n = n < 0 ? -n : n; // strip of sign
    number = new char[BIT_INCREMENT];
    for (int i = 0; i < BIT_INCREMENT; i++)
        {
        number[i] = n & 1;
        n >>= 1;
        }
    max = BIT_INCREMENT - 1;
    sig = BIT_INCREMENT - 1;
    contract(); // remove leading 0s
}

LargeNumber::LargeNumber(const LargeNumber& n)
{
    number = new char[n.max + 1];
    negative = n.negative;
    max = n.max;
    sig = n.sig;
    for (int i = sig; i >= 0; i--)
        number[i] = n.number[i];
}

LargeNumber::~LargeNumber(void)
{
    delete number;
}

char LargeNumber::to_char(void) const
{
    return to_long();
}

short LargeNumber::to_short(void) const
{
    return to_long();
}

int LargeNumber::to_int(void) const
{
    return to_long();
}

long LargeNumber::to_long(void) const
{
    long n = 0;

    for (int i = sig; i >= 0; i--)
        {
        n <<= 1;
        n |= number[i];
        }
    if (negative)
        return -n;
    else
        return n;
}

int LargeNumber::even(void) const
{
    return number[0] == 0;
}
```

```
int LargeNumber::odd(void) const
{
    return number[0] == 1;
}

int LargeNumber::length(void) const
{
    return sig + 1;
}

int LargeNumber::bit(unsigned int p) const
{
    if (p <= sig) // check if within current size
        return number[p];
    else
        return false;
}

int LargeNumber::zero(void) const
{
    return (sig == 0 && number[0] == 0);
}

int LargeNumber::sign(void) const
{
    return negative;
}

void LargeNumber::truncate(int n)
{
    if (n < 1) // either set to zero
        {
        sig = 0;
        number[0] = 0;
        negative = false;
        }
    else if (sig > n - 1) // or chop down
        {
        sig = n - 1;
        contract(); // may have revealed leading zeros
        }
}

void LargeNumber::complement(void)
{
    if (!zero()) // can't have negative zeros
        negative = !negative;
}

int LargeNumber::operator==(const LargeNumber& n) const
{
    if (sig != n.sig) // check size
        return false;
    if (negative != n.negative) // check sign
        return false;
    for (int i = sig; i >= 0; i--)
        if (number[i] != n.number[i]) // check bits
            return false;
    return true;
}

int LargeNumber::operator!=(const LargeNumber& n) const
{
    return !(*this == n);
}
```

```
int LargeNumber::smaller(const LargeNumber& n) const
{
    if (sig < n.sig) // check size
        return true;
    else if (sig > n.sig) // check sign
        return false;
    for (int i = sig; i >= 0; i--)
        if (number[i] < n.number[i]) // check bits
            return true;
        else if (number[i] > n.number[i])
            return false;
    return false;
}

int LargeNumber::greater(const LargeNumber& n) const
{
    if (sig > n.sig) // check size
        return true;
    else if (sig < n.sig) // check sign
        return false;
    for (int i = sig; i >= 0; i--)
        if (number[i] > n.number[i]) // check bits
            return true;
        else if (number[i] < n.number[i])
            return false;
    return false;
}

int LargeNumber::operator<(const LargeNumber& n) const
{
    if (negative & !n.negative) // try to make judgement using signs
        return true;
    else if (!negative & n.negative)
        return false;
    else if (negative)
        return !smaller(n);
    else
        return smaller(n);
}

int LargeNumber::operator<=(const LargeNumber& n) const
{
    return *this < n || *this == n;
}

int LargeNumber::operator>(const LargeNumber& n) const
{
    return !(*this <= n);
}

int LargeNumber::operator>=(const LargeNumber& n) const
{
    return !(*this < n);
}

void LargeNumber::expand(int n)
{
    if (n < sig) // don't need to expand
        return;
    if (max < n) // need to make a larger array
        {
        char* new_number = new char[n + 1];
        for (int i = sig; i >= 0; i--)
            new_number[i] = number[i];
        delete number;
```

```
        number = new_number;
        max = n;
        }
    for (int i = sig + 1; i <= max; i++) // zero top of array
        number[i] = 0;
    sig = n;
}

LargeNumber& LargeNumber::operator=(const LargeNumber& n)
{
    if (this == &n) // same object
        return *this;
    expand(n.sig); // make equal sizes
    sig = n.sig; // might have been larger
    for (int i = sig; i >= 0; i--)
        number[i] = n.number[i];
    negative = n.negative;
    return *this;
}

void LargeNumber::plus(const LargeNumber& n)
{
    int m = maximum(sig + 1, n.sig + 1);
    expand(m); // allow for overflow
    int i = 0;
    int carry = 0;
    for (; i <= n.sig; i++) // add overlap
        {
        carry += number[i] + n.number[i];
        number[i] = carry & 1;
        carry /= 2;
        }
    for (; carry != 0; i++) // continue with carry
        {
        carry += number[i];
        number[i] = carry & 1;
        carry /= 2;
        }
    contract();
}

void LargeNumber::minus(const LargeNumber& n)
{
    int m = maximum(sig, n.sig);
    expand(m);
    int i = 0;
    int carry = 0;
    for (; i <= n.sig; i++) // subtract overflow
        {
        carry += number[i] - n.number[i];
        number[i] = (carry + 2) & 1;
        carry = carry < 0 ? -1 : 0;
        }
    for (; carry != 0; i++) // continue with carry
        {
        carry += number[i];
        number[i] = (carry + 2) & 1;
        carry = carry < 0 ? -1 : 0;
        }
    contract();
}

LargeNumber& LargeNumber::operator+=(const LargeNumber& n)
{
    if ((negative ^ n.negative) == false) // cope with negatives
        this->plus(n);
```

```
    else
        {
        if (smaller(n))
            {
            LargeNumber m(*this);
            *this = n;
            minus(m);
            }
        else
            this->minus(n);
        if (zero())
            negative = false;
        }
    return *this;
}

LargeNumber& LargeNumber::operator-=(const LargeNumber& n)
{
    if ((negative ^ n.negative) == true) // cope with negatives
        this->plus(n);
    else
        {
        if (smaller(n))
            {
            LargeNumber m(*this);
            *this = n;
            minus(m);
            complement();
            }
        else
            this->minus(n);
        if (zero())
            negative = false;
        }
    return *this;
}

LargeNumber& LargeNumber::operator<<=(int n)
{
    if (n < 0) // avoid negatives
        {
        *this >>= -n;
        return *this;
        }
    expand(sig + n);
    for (int i = sig; i >= n; i--) // copy
        number[i] = number[i - n];
    for (i = n - 1; i >= 0; i--) // then fill with 0s
        number[i] = 0;
    contract();
    return *this;
}

LargeNumber& LargeNumber::operator>>=(int n)
{
    if (n < 0) // avoid negatives
        {
        *this <<= -n;
        return *this;
        }
    // why can't I use - for (int i = 0; i <= sig - n; i++)
    for (int i = 0; i <= sig; i++) // copy
        number[i] = number[i + n];
    sig = sig - n; // shorten
    if (sig < 0)
        {
```

```
        sig = 0;
        number[0] = 0;
        }
    if (zero())
        negative = false;
    return *this;
}

LargeNumber& LargeNumber::operator++(void)
{
    return (*this += 1);
}

LargeNumber& LargeNumber::operator--(void)
{
    return (*this -= 1);
}

LargeNumber LargeNumber::operator++(int)
{
    LargeNumber c = *this;
    *this += 1;
    return c;
}

LargeNumber LargeNumber::operator--(int)
{
    LargeNumber c = *this;
    *this -= 1;
    return c;
}

LargeNumber& LargeNumber::operator*=(const LargeNumber& n)
{
    LargeNumber c;
    int m = sig + n.sig;
    expand(m);
    if (n.smaller(*this)) // loop through the smaller number
        {
        for (int i = 0; i <= n.sig; i++)
            {
            if (n.number[i] == 1)
                c.plus(*this); // add on multiples of two
            *this <<= 1;
            }
        }
    else
        {
        LargeNumber m = n;
        for (int i = 0; i <= sig; i++)
            {
            if (number[i] == 1)
                c.plus(m); // add on multiples of two
            m <<= 1;
            }
        }
    if (c.zero()) // check negatives
        c.negative = false;
    else
        c.negative = negative ^ n.negative;
    *this = c;
    contract();
    return *this;
}
```

```
/*
  LargeNumber& LargeNumber::operator/=(const LargeNumber& n)

  Modified by Brice Canvel in June 1999 as a bug was found in this function
  */

LargeNumber& LargeNumber::operator/=(const LargeNumber& n)
{
    LargeNumber temp;
    LargeNumber c;
    LargeNumber m = n;
    if (n.zero()) // no divide by zero
          throw;
    m <<= maximum(sig - n.sig, 0); // power of two multiple of n

    /*
      The power function used here was before the function pow
      defined earlier in the class which takes two integers to type long
      and returns an integer if type long.
      The number obtained though were outside the range of numbers of type
      long causing the function /= to give wrong results.
      The new function power uses numbers of type LargeNumber which resolves the
      problem.
      */

    for (LargeNumber i = power(2, sig - n.sig); i > 0; i >>= 1)
        {
        if (!m.greater(*this))
            {
            minus(m); // subtract of large chunk at time
            c += i;
            }
        m >>= 1; // shrink chunk down
        }
    if (c.zero()) // check negatives
        c.negative = false;
    else
        c.negative = negative ^ n.negative;
    *this = c;
    return *this;
}

LargeNumber& LargeNumber::operator%=(const LargeNumber& n)
{
    LargeNumber m = n;
    if (n.zero()) // no divide by zero
          throw;
    m <<= maximum(sig - n.sig, 0); // power of two multiple of n
    for (int i = sig - n.sig; i >= 0; i--)
        {
        if (!m.greater(*this))
            minus(m); // subtract of large chunk at time
        m >>= 1; // shrink chunk down
        }
    if (zero())
        negative = false;
    return *this;
}

LargeNumber& LargeNumber::operator&=(const LargeNumber& n)
{
    int m = maximum(sig, n.sig);
    expand(m); // match sizes
    for (int i = minimum(sig, n.sig); i >= 0; i--)
        number[i] &= n.number[i];
    contract();
```

```
    return *this;
}

LargeNumber& LargeNumber::operator|=(const LargeNumber& n)
{
    int m = maximum(sig, n.sig);
    expand(m); // match sizes
    for (int i = minimum(sig, n.sig); i >= 0; i--)
        number[i] |= n.number[i];
    contract();
    return *this;
}

LargeNumber& LargeNumber::operator^=(const LargeNumber& n)
{
    int m = maximum(sig, n.sig);
    expand(m); // match sizes
    for (int i = minimum(sig, n.sig); i >= 0; i--)
        number[i] ^= n.number[i];
    contract();
    return *this;
}

LargeNumber LargeNumber::operator+(const LargeNumber& n) const
{
    LargeNumber c = *this;
    c += n;
    return c;
}

LargeNumber LargeNumber::operator-(const LargeNumber& n) const
{
    LargeNumber c = *this;
    c -= n;
    return c;
}

LargeNumber LargeNumber::operator*(const LargeNumber& n) const
{
    LargeNumber c = *this;
    c *= n;
    return c;
}

LargeNumber LargeNumber::operator/(const LargeNumber& n) const
{
    LargeNumber c = *this;
    c /= n;
    return c;
}

LargeNumber LargeNumber::operator%(const LargeNumber& n) const
{
    LargeNumber c = *this;
    c %= n;
    return c;
}

LargeNumber LargeNumber::operator&(const LargeNumber& n) const
{
    LargeNumber c = *this;
    c &= n;
    return c;
}

LargeNumber LargeNumber::operator|(const LargeNumber& n) const
```

```
{
    LargeNumber c = *this;
    c |= n;
    return c;
}

LargeNumber LargeNumber::operator^(const LargeNumber& n) const
{
    LargeNumber c = *this;
    c ^= n;
    return c;
}

LargeNumber LargeNumber::operator<<(int n) const
{
    LargeNumber c = *this;
    c <<= n;
    return c;
}

LargeNumber LargeNumber::operator>>(int n) const
{
    LargeNumber c = *this;
    c >>= n;
    return c;
}

ostream& operator<<(ostream& s, const LargeNumber& n)
{
    if (n.negative)
        s << '-';
    for (int i = n.sig; i >= 0; i--)
        s << char(n.number[i] + '0');
    return s;
}

istream& operator>>(istream& s, LargeNumber& n)
{
    char c;
    while (s.get(c)) // strip any leading spaces
        if (c != ' ' && c != '\n' && c != '\r')
            {
            s.putback(c);
            break;
            }
    n = 0;
    while (s.get(c)) // check for negative
        {
        if (c != '-' & c != '+')
            {
            s.putback(c);
            break;
            }
        if (c == '-')
            n.negative = !n.negative;
        }
    while (s.get(c)) // build digits in reverse
        {
        if (c != '0' & c != '1')
            {
            s.putback(c);
            break;
            }
        if (n.sig > n.max) {
            n.expand(n.sig + BIT_INCREMENT);
            n.sig -= BIT_INCREMENT;
```

```
            }
        n.number[n.sig++] = c - '0';
        }
    if (n.sig > 0) // put digits right way round
        {
        n.sig--;
        for (int j = n.sig; j > n.sig / 2; j--)
            {
            c = n.number[j];
            n.number[j] = n.number[n.sig - j];
            n.number[n.sig - j] = c;
            }
        n.contract();
        }
    return s;
}

/*******************************************************************/
/*   Author : Brice Canvel                                         */
/*******************************************************************/
/*   Purpose : Implementation of the RSA algorithm                 */
/*             and related functions using the class               */
/*             LargeNumber as a representation of very large       */
/*             integers                                            */
/*******************************************************************/



/*******************************************************************/
/*   G  C  D                                                       */
/*******************************************************************/

/*
  LargeNumber gcd ( const LargeNumber &X, const LargeNumber &Y ) :

  Finds the greatest common divisor to two numbers X and Y

  */

LargeNumber gcd( const LargeNumber &X, const LargeNumber &Y )
{
  LargeNumber x=X, y=Y;
  while (1)
  {
    if ( y == 0 ) return x;
    x = x % y;
    if ( x == 0 ) return y;
    y = y % x;
  }
}

/*******************************************************************/
/*   M O D I N V                                                   */
/*******************************************************************/

/*
  LargeNumber modinv ( const LargeNumber &a, const LargeNumber &m ) :

  Finds the inverse of a number a modulo m

  */

LargeNumber modinv( const LargeNumber &a, const LargeNumber &m ) // modular inverse
// returns i in range 1..m-1 such that i*a = 1 mod m
// a must be in range 1..m-1
```

```
{
  LargeNumber j=1,i=0,b=m,c=a,x,y;
  while ( c > 0 )
  {
    x = b / c;
    y = b - x*c;
    b = c;
    c = y;
    y = j;
    j = i - j*x;
    i = y;
  }
  if ( i < 0 )
    i += m;
  return i;
}

/*****************************************************************/
/*   S Q R T                                                     */
/*****************************************************************/

/* LargeNumber Sqrt ( LargeNumber n ) :

   Calculates the nearest integer value below the square root of a number n
   eg SQRT(1234)=35

   */

LargeNumber Sqrt(LargeNumber n)
{
  LargeNumber res=n,old_res,l_res,r_res,temp;

  while( (res*res) > n)
    {
      old_res=res;
      res>>=1;
    }

  l_res=res;
  r_res=old_res;

  while (l_res != (r_res-1) )
    {
      res=(l_res+r_res)>>1;
      if ( (res*res) < n )
l_res=res;
      else
r_res=res;
    }

  return l_res;
}

/*****************************************************************/
/*   P O W E R                                                   */
/*****************************************************************/

/*
  LargeNumber power ( LargeNumber x, LargeNumber n ) :

  Calculates x to the power n where x and n are binary numbers

  */

LargeNumber power ( LargeNumber x, LargeNumber n )
{
```

```
  LargeNumber A;

  A=1;

  for (int i=n.length()-1;i>=0;i--)
    {
      A=A*A;
      if (n.bit(i) == 1)
A=A*x;
    }

  return A;
}

/*****************************************************************/
/*   M U L T I P L Y M O D N                                      */
/*****************************************************************/

/*
  LargeNumber MultiplyModN (LargeNumber x,LargeNumber y,LargeNumber m,LargeNumber sqrt_m) :

  Calculates the product x*y modulo m using Head's algorithm

  */

LargeNumber MultiplyModN (LargeNumber x,LargeNumber y,LargeNumber m,LargeNumber sqrt_m)
{
  LargeNumber a,b,c,d,z,e,f,v,v1,g,h,j,j1,k,capT,t,prod;

  capT=sqrt_m;
  t=capT*capT-m;

  while ( (capT*capT) > LargeNumber(2)*m || t > capT || t < LargeNumber(-1)*capT )
    {
      capT=capT+1;
      t=capT*capT-m;
    }

  a=x/capT; b=x-a*capT;
  c=y/capT; d=y-c*capT;
  z=a*d+b*c;
  z=z-m*(z/m);
  e=(a*c)/capT; f=a*c-e*capT;
  v=z+e*t;
  v1=v/m;
  if ( (v>0) || ( (v % m) == 0 ) )
    v=v-m*v1;
  else
    v=v-m*(v1-1);
  g=v/capT; h=v-g*capT;
  j=(f+g)*t;
  j1=j/m;
  if ( (j>0) || ( (j % m) == 0 ) )
    j=j-m*j1;
  else
    j=j-m*(j1-1);
  k=j+b*d;
  k=k-m*(k/m);
  prod=h*capT+k;
  prod=prod-m*(prod/m);
  if (prod<0)
    prod+=m;

  return prod;
}
```

```
/******************************************************************/
/*   P O W E R R U L E                                            */
/******************************************************************/

/*
  LargeNumber PowerRule ( LargeNumber x, LargeNumber n, LargeNumber m ) :

  Calculates x to the power n modulo m using the function MultiplyModN

  */

LargeNumber PowerRule ( LargeNumber x, LargeNumber n, LargeNumber m, double *t )
{
  LargeNumber e,res,two=2,sqrt_m;
  Timer test;
  int i=0,size=n.length();

  res=1;

  sqrt_m=Sqrt(m);

  while ( n > 0 )
    {
      e=n-two*(n>>1);
      test.start();
      if ( e == 1)
res=MultiplyModN(x,res,m,sqrt_m);
      x=MultiplyModN(x,x,m,sqrt_m);
      test.stop();
      *(t+i)=test.cpuTime();
      n=(n-e);
      n>>=1;
      i++;
    }

  return res;
}

/******************************************************************/
/*   I S _ P R I M E                                              */
/******************************************************************/

/*
  LargeNumber is_prime ( LargeNumber testNum ) :

  Checks if the number testNum is prime

  */

LargeNumber is_prime ( LargeNumber testNum )
{
  int prime;
  LargeNumber stop,trialDiv;

  if ( (testNum == 0) || (testNum == 1) )
    prime=false;
  else
    prime=true;
  trialDiv=2;
  stop=Sqrt(testNum);
  while ( (trialDiv <= stop) && prime )
    {
      if ( (testNum % trialDiv) == 0 )
prime=false;
      trialDiv++;
    }
```

```
  return prime;
}

/*****************************************************************/
/*   O D D R A N D O M                                           */
/*****************************************************************/

/*
  long OddRandom ( long bit_length ) :

  Finds a random prime number of length bit_length bits

  */

long OddRandom ( long bit_length )
{
  long i,mask = 1, n;

  bit_length--;
  for (i = 1; i <= bit_length; i++)
    mask |= 1 << i;
  if (bit_length < 16)
    n = rand();
  else
    n = (rand() << 16) | rand();
  n |= (1 << bit_length);
  n &= mask;
  if ((n & 1) == 0) n++;
  return n;
}
```

A.4 timer.h

```
#ifndef TIMER_H
#define TIMER_H

/*
   C++ interface/implementation of a Timer class

   July 30 1996, Roque D. Oliveira, U. Pennsylvania, roque@penn01.fnal.gov

*/

#include <iostream.h>
#include <unistd.h>     /* _SC_CLK_TCK */
#include <sys/types.h>  /* clock_t     */
#include <sys/times.h>  /* times()     */
#include <time.h>       /* clock()     */

// The CPUTimer class keeps track of the amount of CPU time used, in seconds.
// The time reported by cpuTime() is the sum of the CPU time
// of the calling process and its terminated child processes.
// It has a high resolution (determined by CLOCKS_PER_SEC, which is
// usually equal to 1000000; see /usr/include/time.h).
//
class CPUTimer
{
  public:
          CPUTimer() { cpuTime(); } // constructor

          double cpuTime()
          {
            clock_t currentTime( clock() );
            double d = ( (double) ( currentTime - _previousTime )) / CLOCKS_PER_SEC;
            _previousTime = currentTime;
            return d;
          }

  private:
          clock_t _previousTime;
};

// The Timer class keeps track of the amount of CPU time used, in seconds,
// as well as its two components (user and system time). It also
// keeps track of the wall clock time.
// It has a low resolution (determined by CLK_TCK, which is
// equal to sysconf(_SC_CLK_TCK), which is usually equal to 100;
// see /usr/include/time.h).
//
//static const long clock_ticks_per_second sysconf (_SC_CLK_TCK);
//
class Timer
{
          inline friend ostream & operator << ( ostream &os, const Timer &t );
  public:
          Timer()  // constructor
          {
             if ((_start_time = times(&_start)) == -1)
            {
                cerr << "Timer::Timer() Error unable to obtain start times" << endl;
                _start_time = 0;
            }
            _finish_time = -1000;
          }

          void   start()
          {
```

```
           if ((_start_time = times(&_start)) == -1)
          {
              cerr << "Timer::start(): Error unable to obtain start times" << endl;
              _start_time = 0;
              return;
          }
          // To prevent erroneous results in case the user doesn't call stop()
          // we also set _finish_time = _start_time  and copy
          // _start.tms_utime onto _finish.tms_utime and copy
          // _start.tms_stime onto _finish.tms_stime
          _finish_time = _start_time;
           _start.tms_utime  += _start.tms_cutime;
          _start.tms_stime  += _start.tms_cstime;
          _finish.tms_utime  = _start.tms_utime;
          _finish.tms_stime  = _start.tms_stime;
        }

        void   stop()
        {
          if ((_finish_time = times(&_finish)) == -1)
          {
              cerr << "\nTimer::stop() Error unable to obtain finish times" << endl;
              _finish_time = -1000;
              return;
          }
          _finish.tms_utime += _finish.tms_cutime;
          _finish.tms_stime += _finish.tms_cstime;
        }

        double userTime()   const
        {
          if ( _finish_time < _start_time )
          {
              cerr << "\nTimer::userTime() Error  _finish_time < _start_time " << endl;
              return 0.0;
          }
          else
              return ((double) (_finish.tms_utime - _start.tms_utime))/CLK_TCK;
        }

        double systemTime() const
        {
          if ( _finish_time < _start_time )
          {
              cerr << "\nTimer::systemTime() Error  _finish_time < _start_time " << endl;
              return 0.0;
          }
          else
              return ((double) (_finish.tms_stime - _start.tms_stime))/CLK_TCK;
        }

        double cpuTime()    const { return userTime() + systemTime(); }

        double wallTime()   const
        {
          if ( _finish_time < _start_time )
          {
              cerr << "\nTimer::wallTime() Error  _finish_time < _start_time " << endl;
              return 0.0;
          }
          else
              return ((double) (_finish_time      - _start_time     ))/CLK_TCK;
        }

private:
        struct tms _start;
```

```
          struct tms _finish;
          clock_t    _start_time;
          clock_t    _finish_time;
};

//inline
ostream& operator << ( ostream &os, const Timer& t )
{
  os << "_start_time = " << t._start_time
     << ", _finish_time = " << t._finish_time << '\n'
     << "_start.tms_utime= " << t._start.tms_utime
     << ", _start.tms_stime= " << t._start.tms_stime << '\n'
     << "_finish.tms_utime= " << t._finish.tms_utime
     << ", _finish.tms_stime= " << t._finish.tms_stime
     << endl;
  return os;
}

#endif
```

Appendix B

Graphs

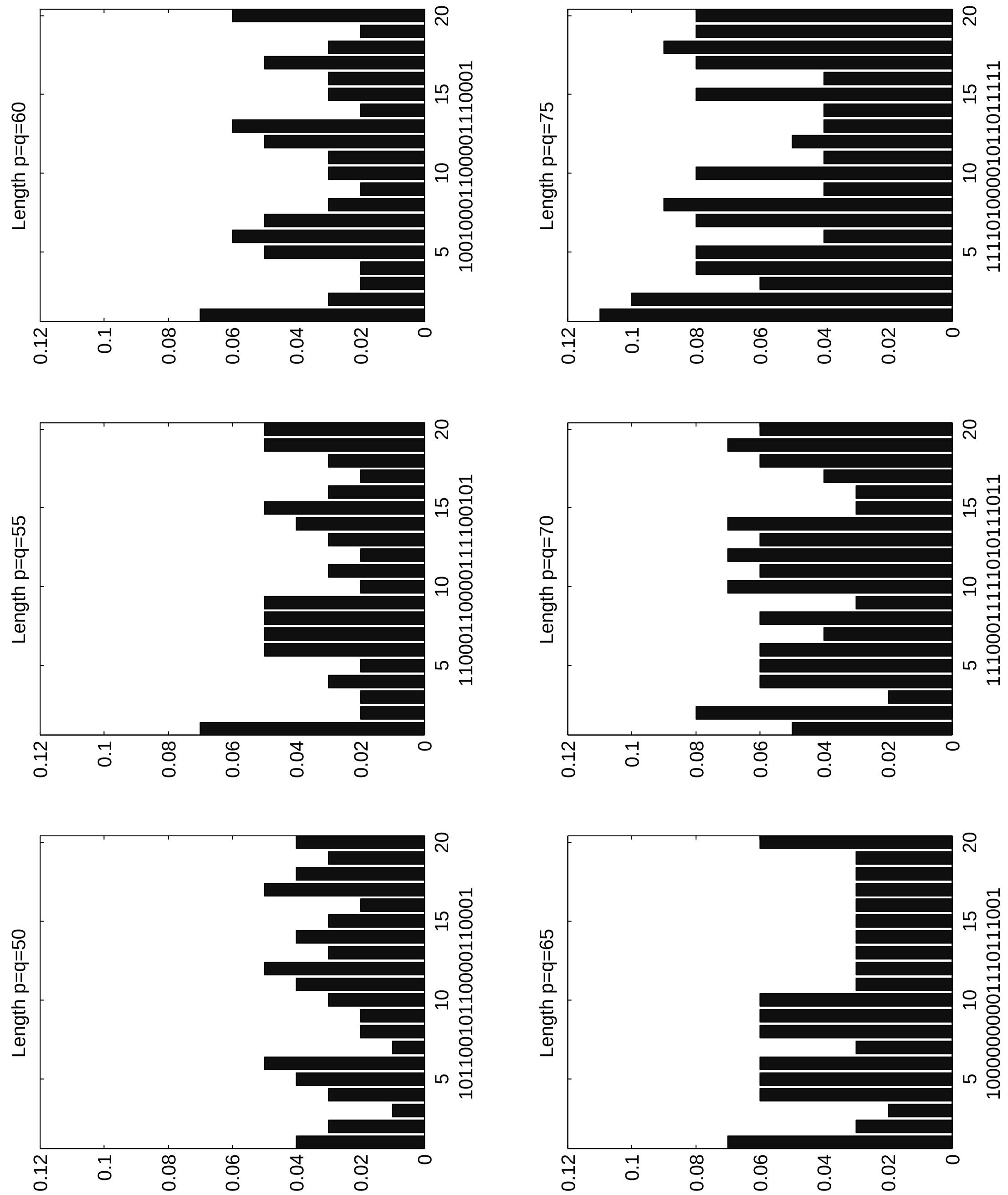


Figure B.1: Timings for an encrypting key of size 20 bits with the size of p and q varying from 50 to 75 bit

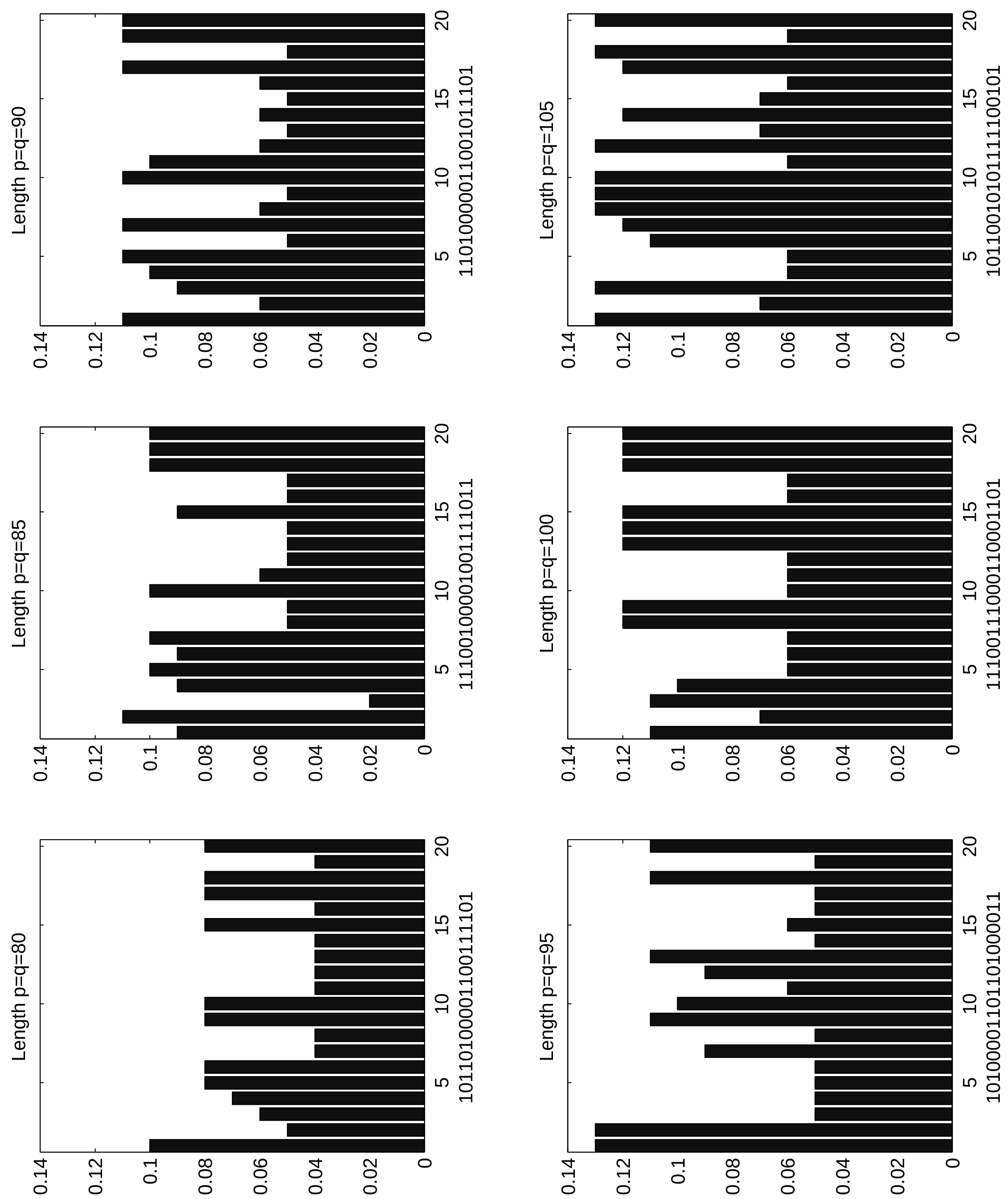


Figure B.2: Timings for an encrypting key of size 20 bit with the size of p and q varying from 80 to 105 bit

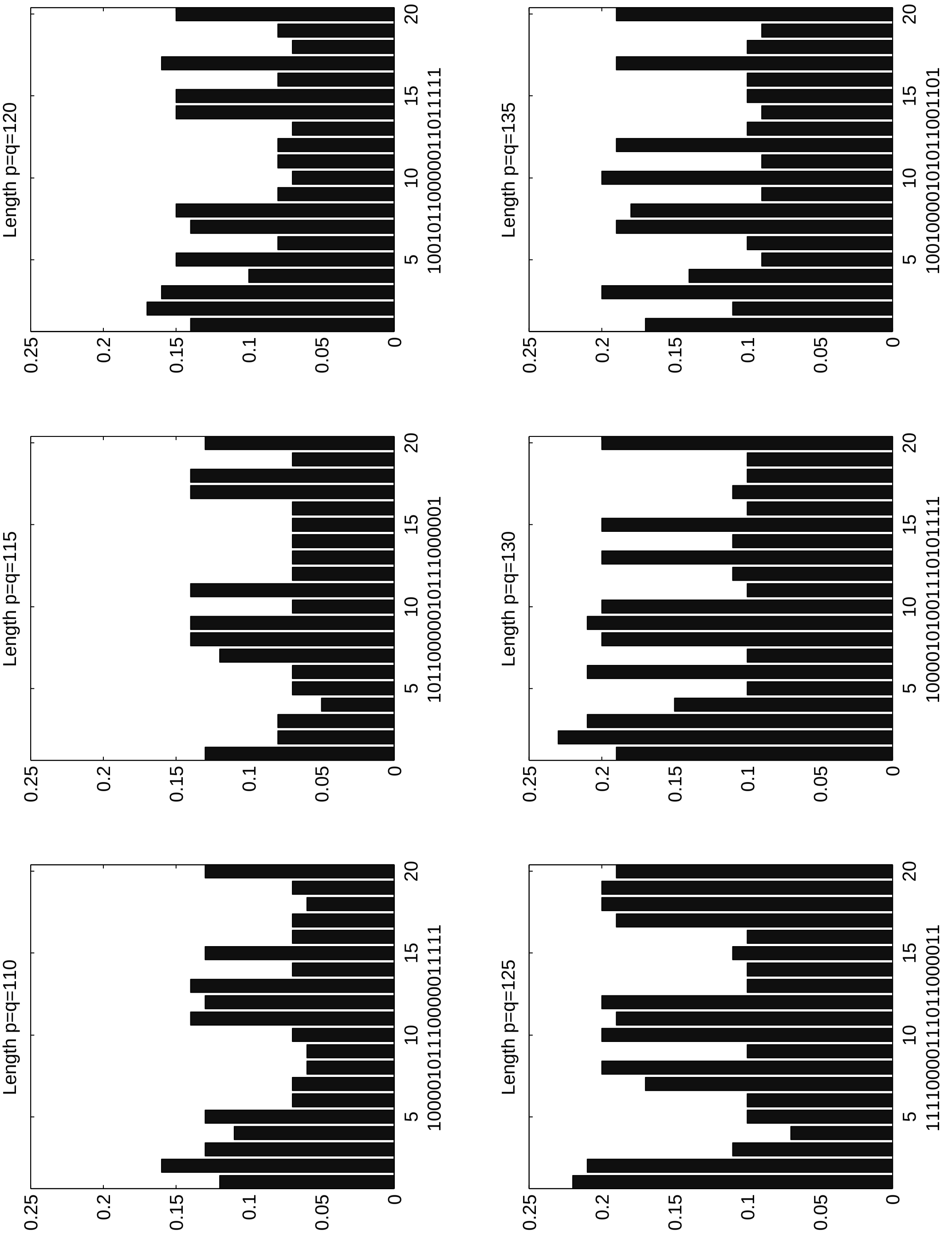


Figure B.3: Timings for an encrypting key of size 20 bit with the size of p and q varying from 110 to 135 bit

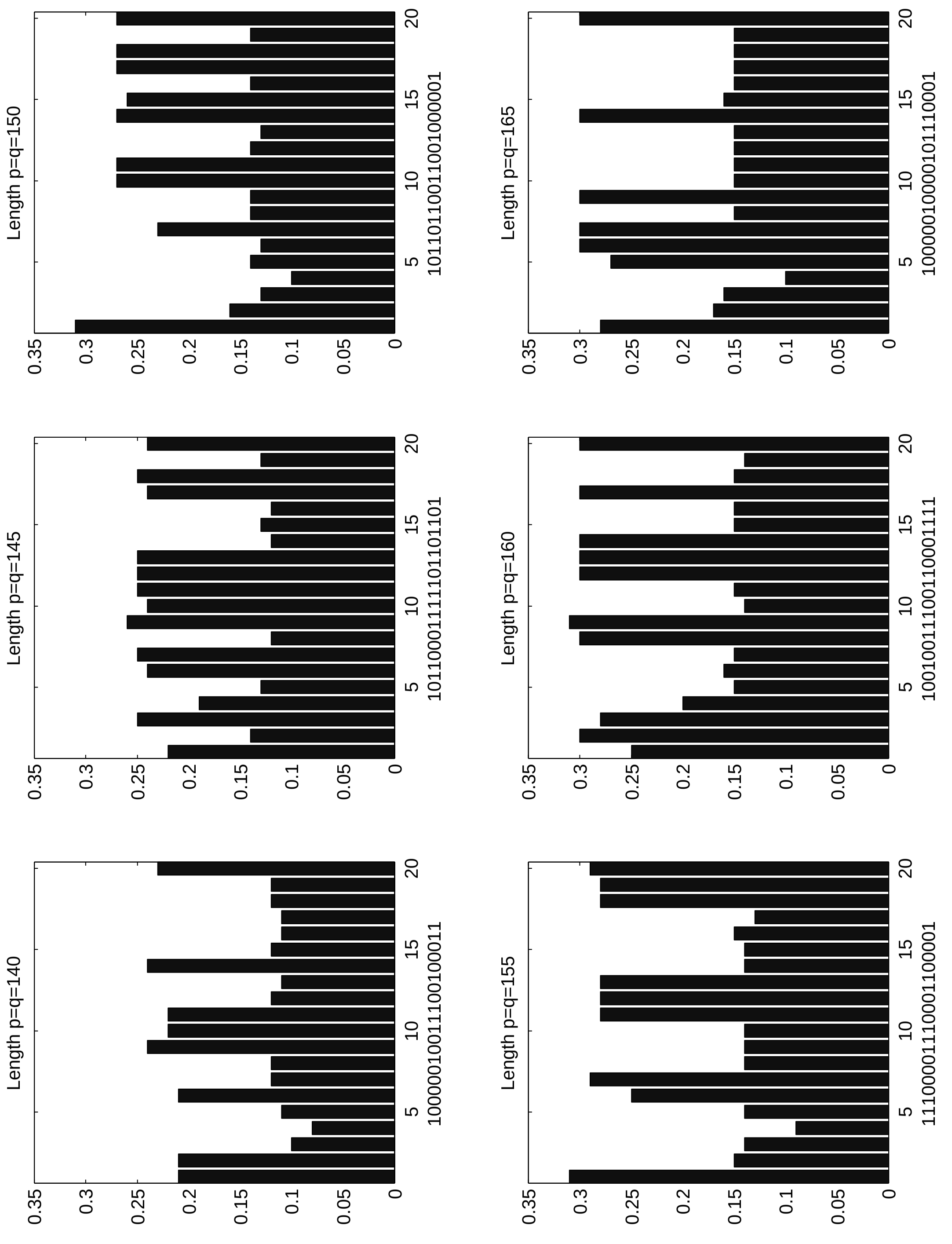


Figure B.4: Timings for an encrypting key of size 20 bit with the size of p and q varying from 140 to 165 bit

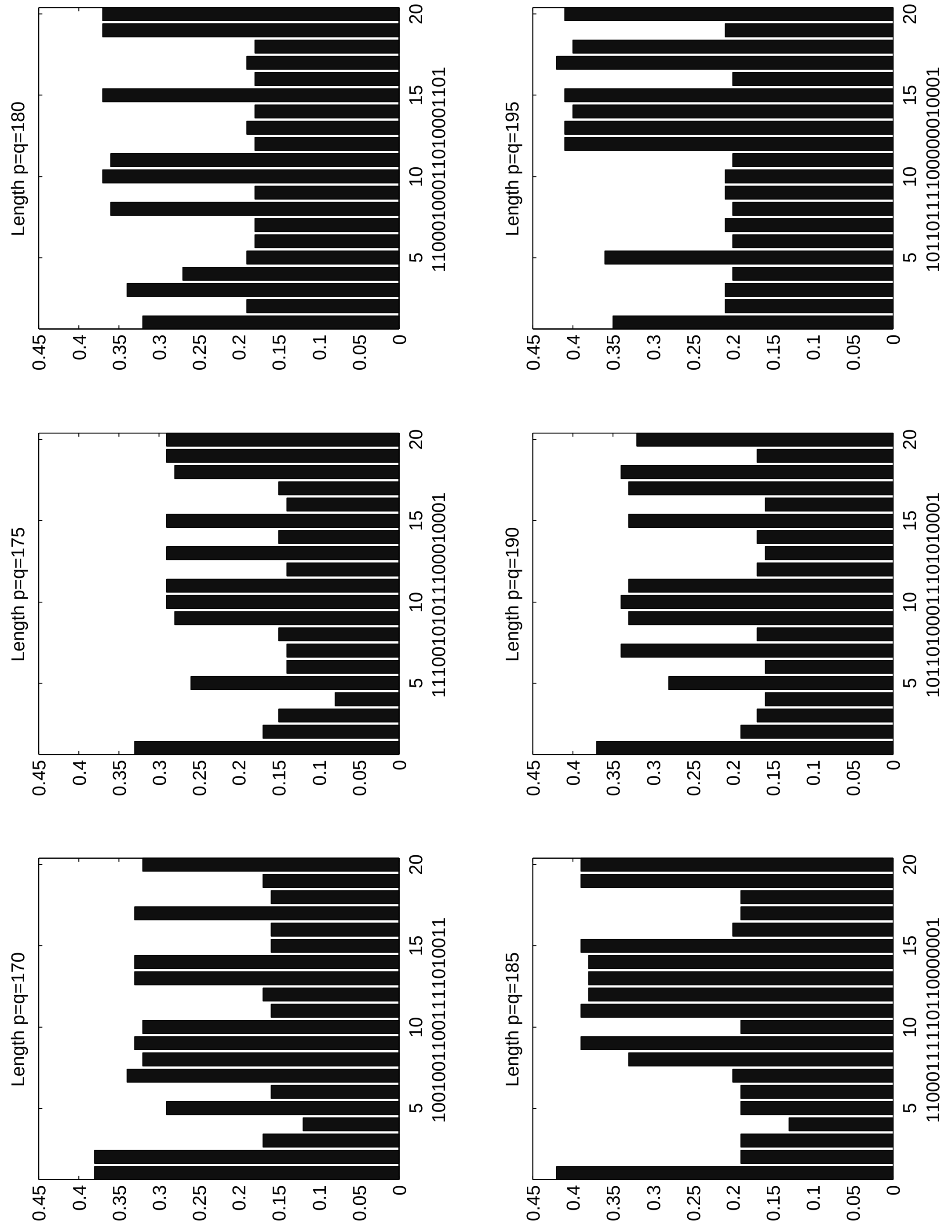


Figure B.5: Timings for an encrypting key of size 20 bit with the size of p and q varying from 170 to 195 bit

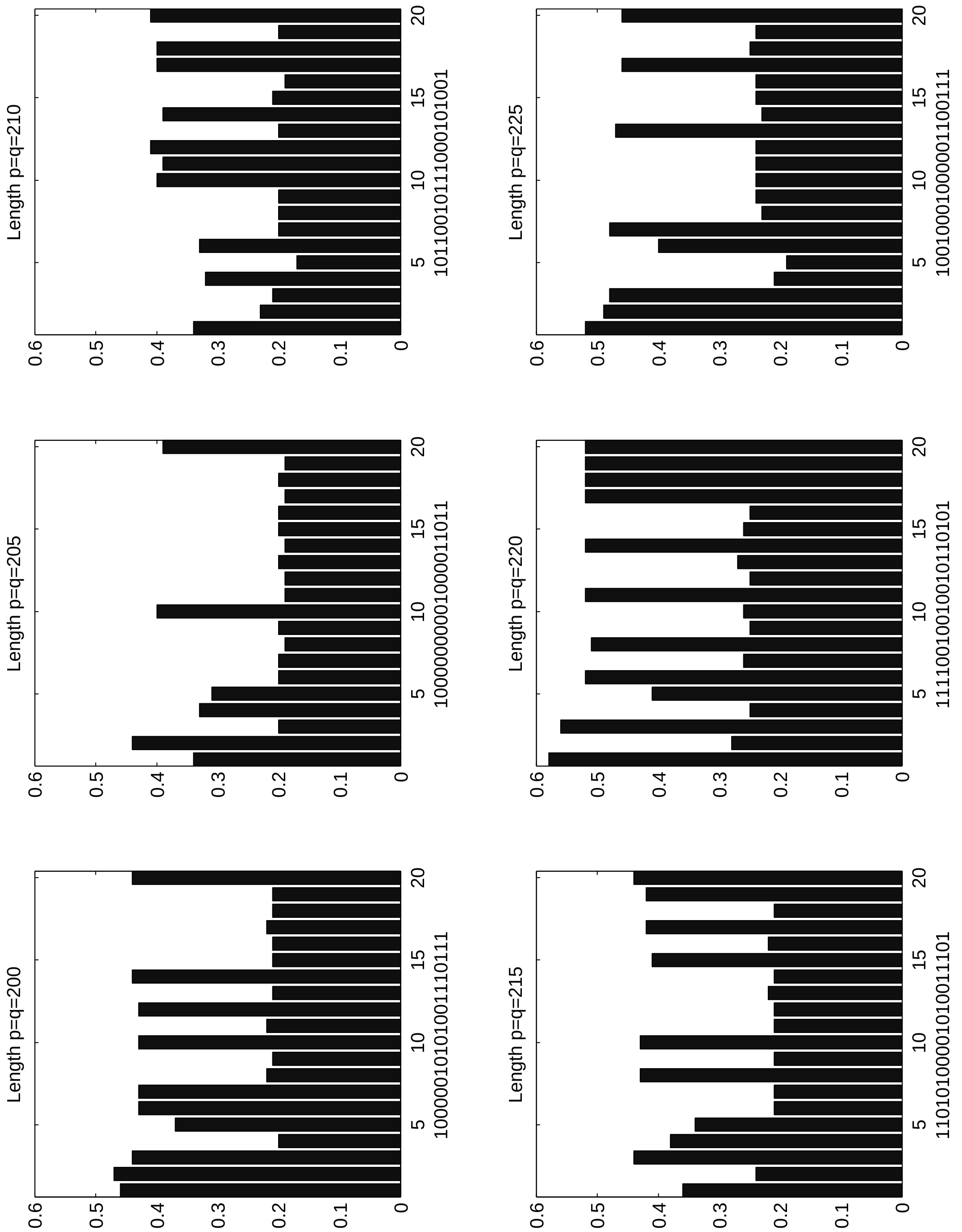


Figure B.6: Timings for an encrypting key of size 20 bit with the size of p and q varying from 200 to 225 bit

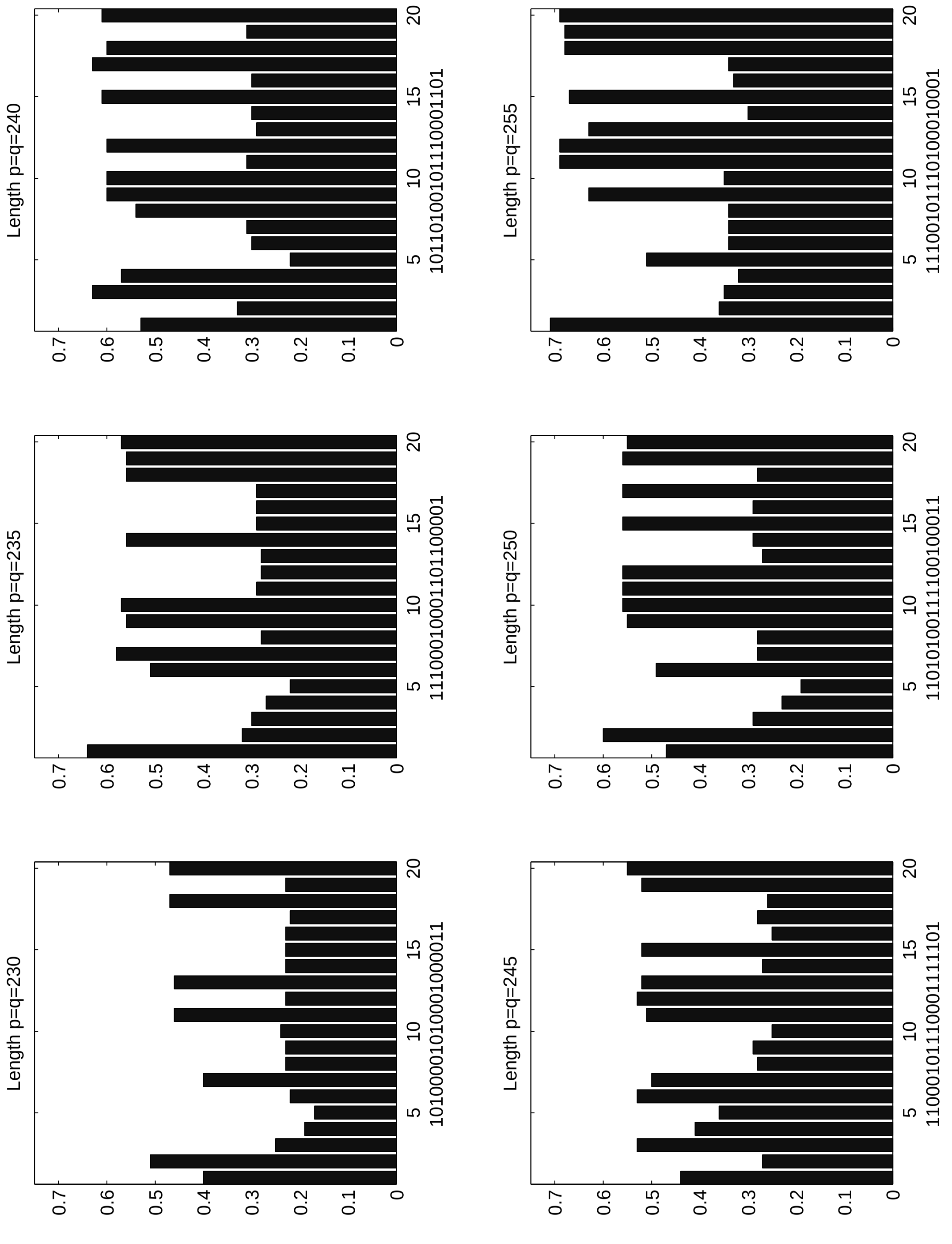


Figure B.7: Timings for an encrypting key of size 20 bit with the size of p and q varying from 230 to 255 bit

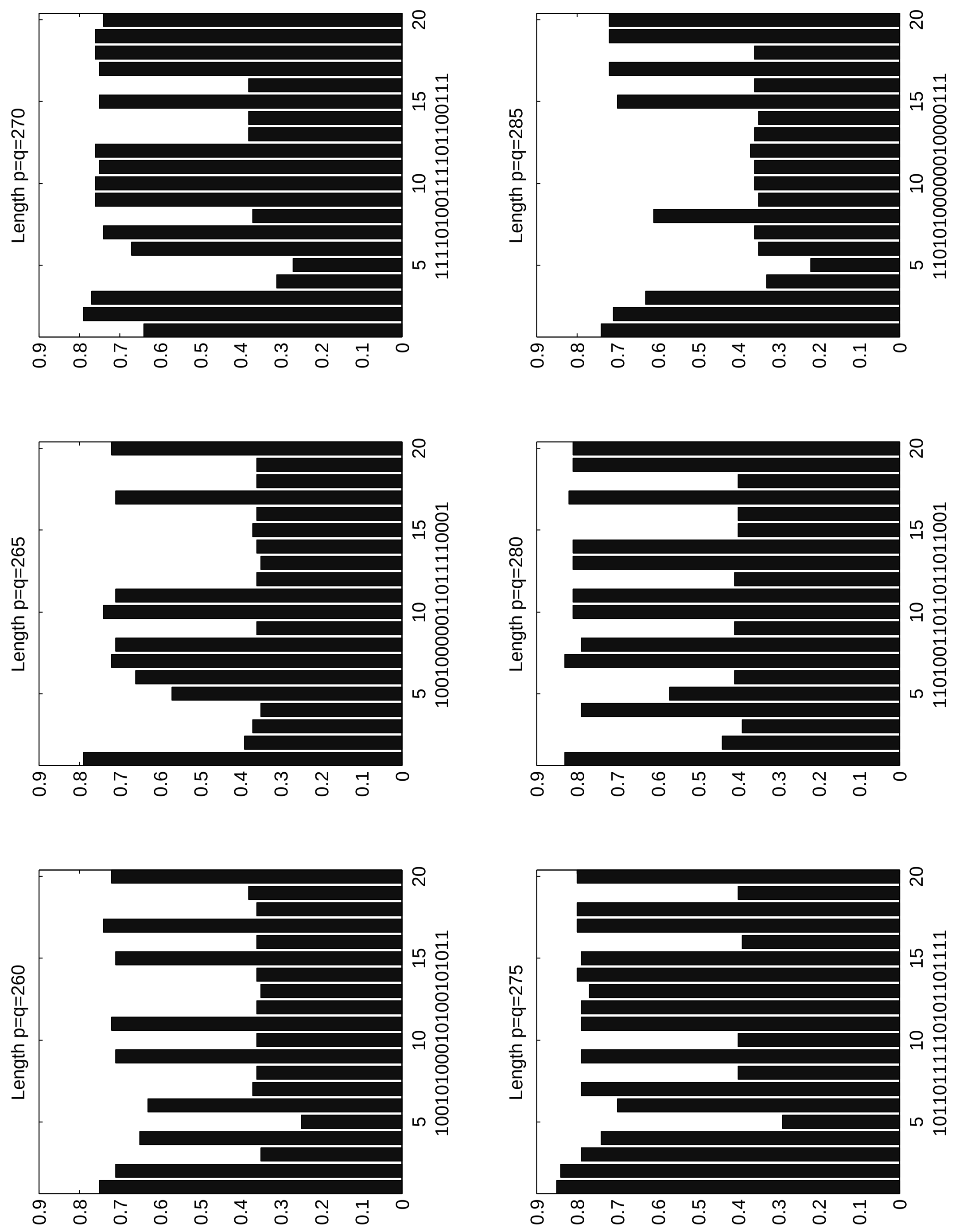


Figure B.8: Timings for an encrypting key of size 20 bit with the size of p and q varying from 260 to 285 bit

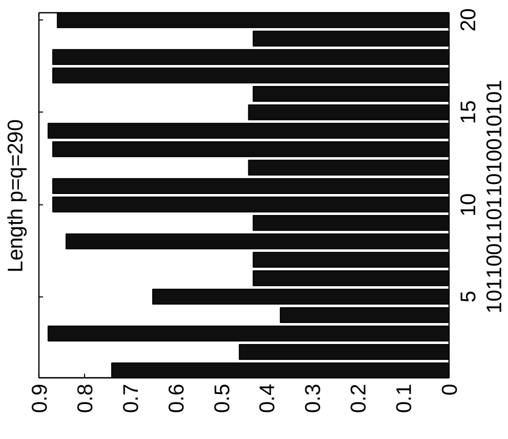


Figure B.9: Timings for an encrypting key of size 20 bit with p and q of size 290 bit

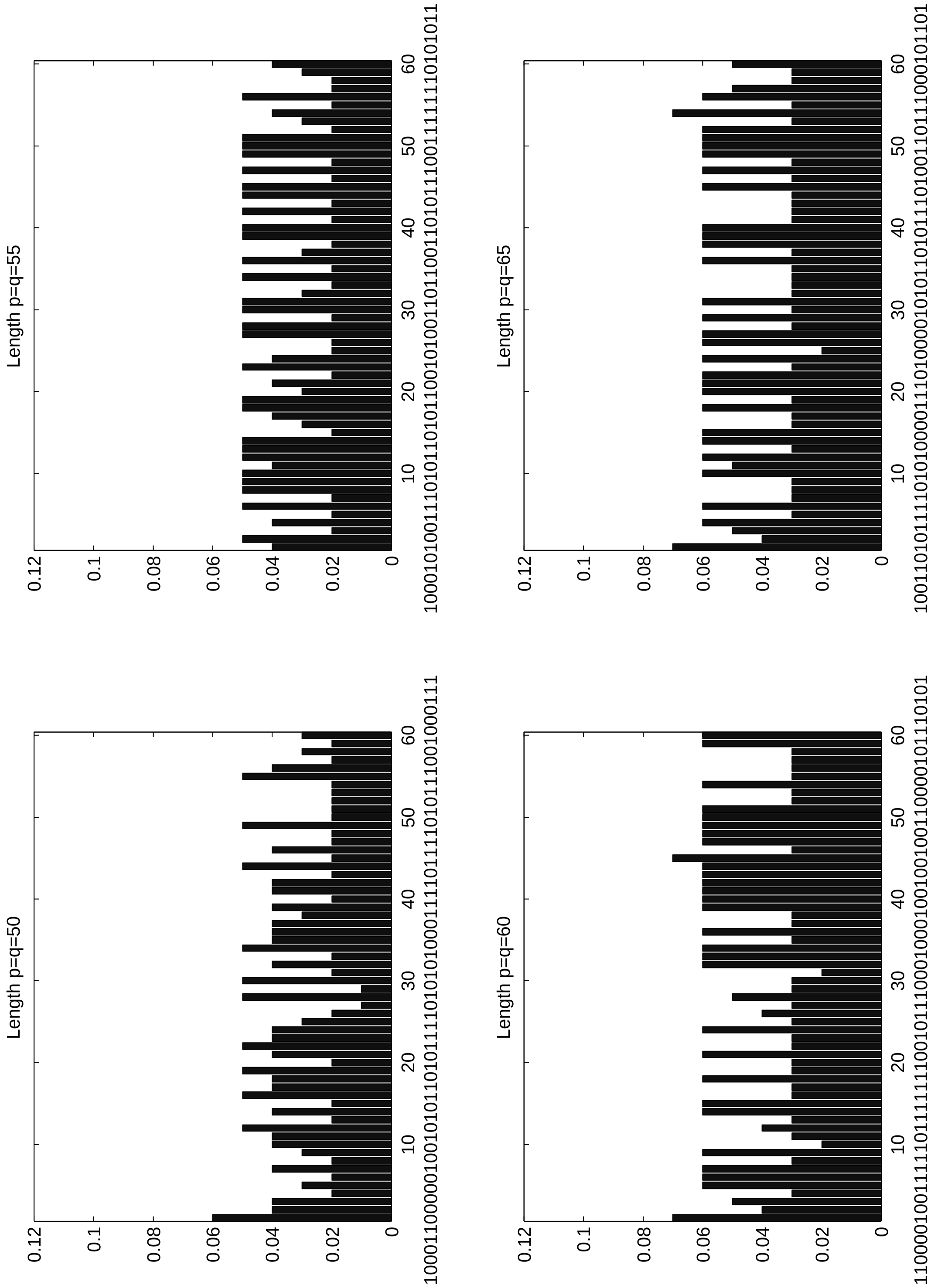


Figure B.10: Timings for an encrypting key of size 60 bit with the size of p and q varying from 50 to 65 bit

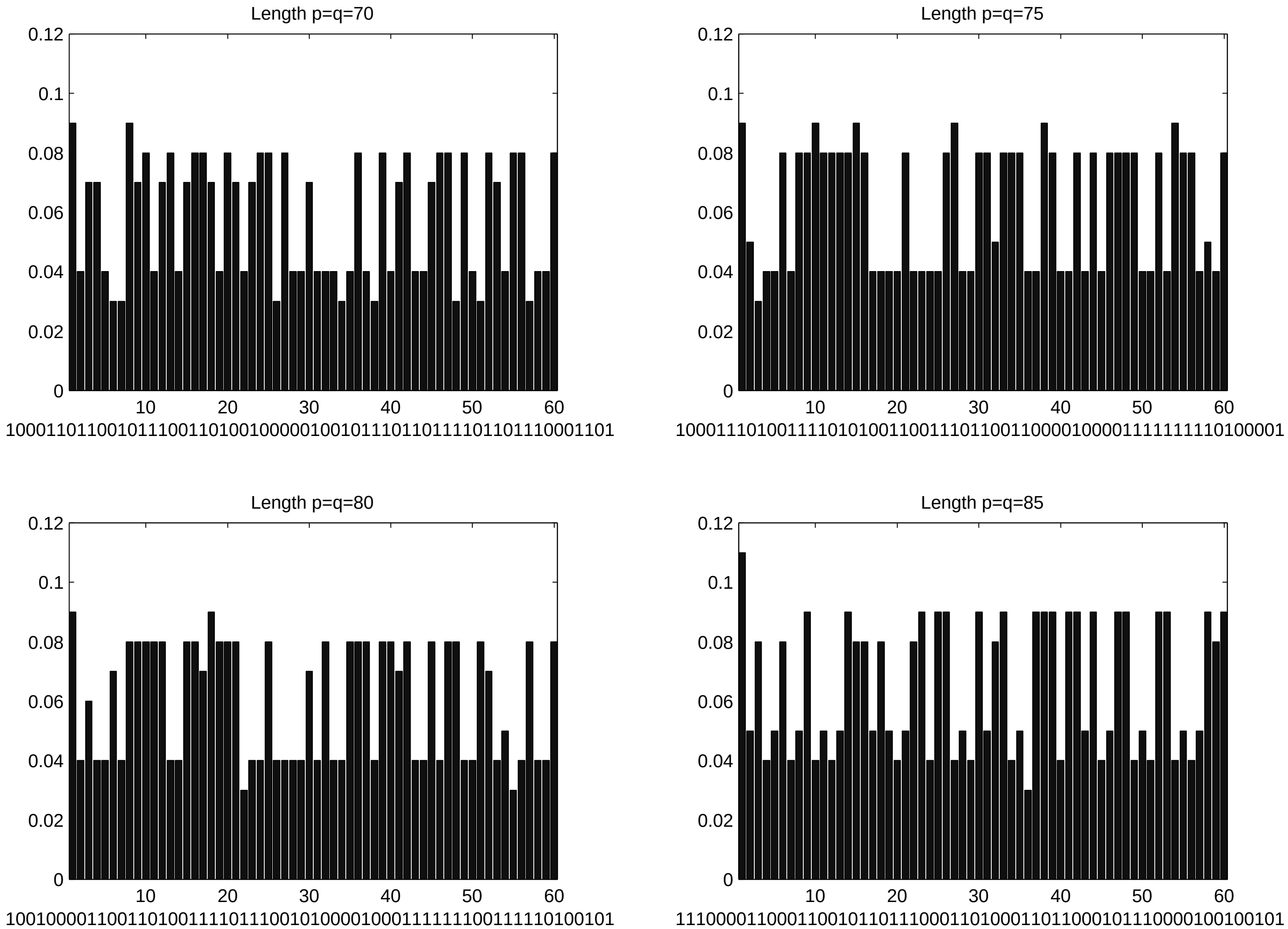


Figure B.11: Timings for an encrypting key of size 60 bit with the size of p and q varying from 70 to 85 bit

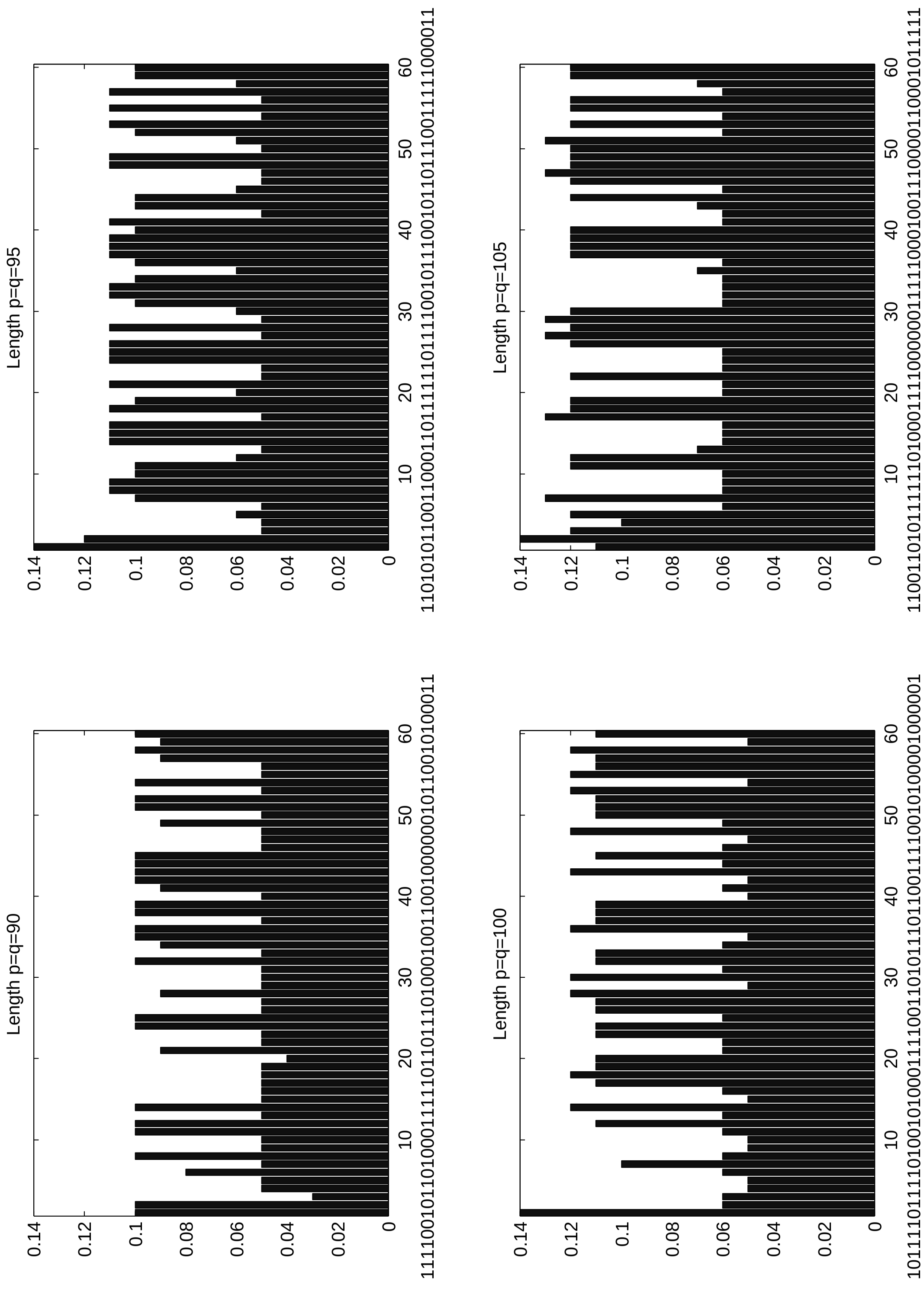


Figure B.12: Timings for an encrypting key of size 60 bit with the size of p and q varying from 90 to 105 bit

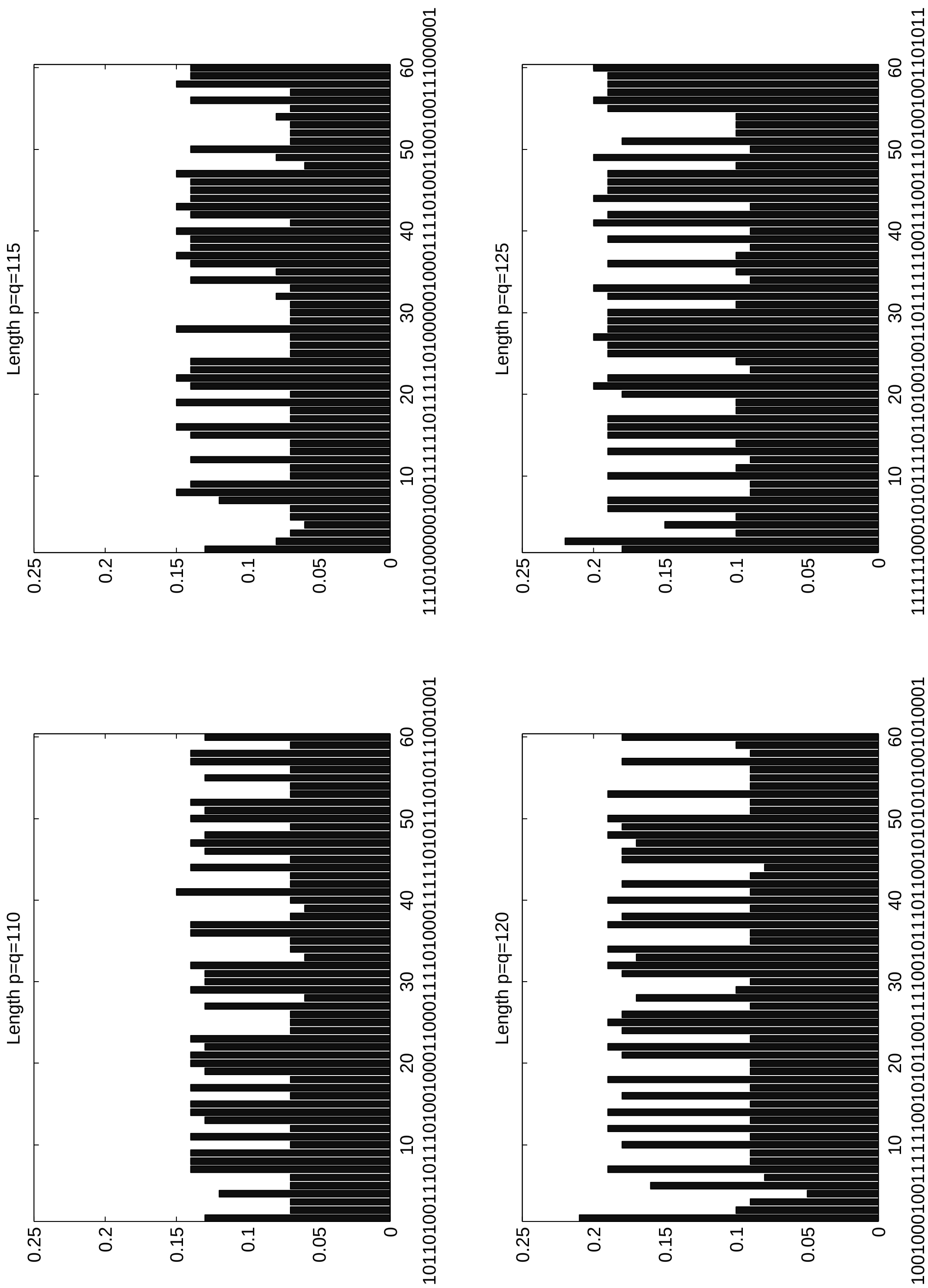


Figure B.13: Timings for an encrypting key of size 60 bit with the size of p and q varying from 110 to 125 bit

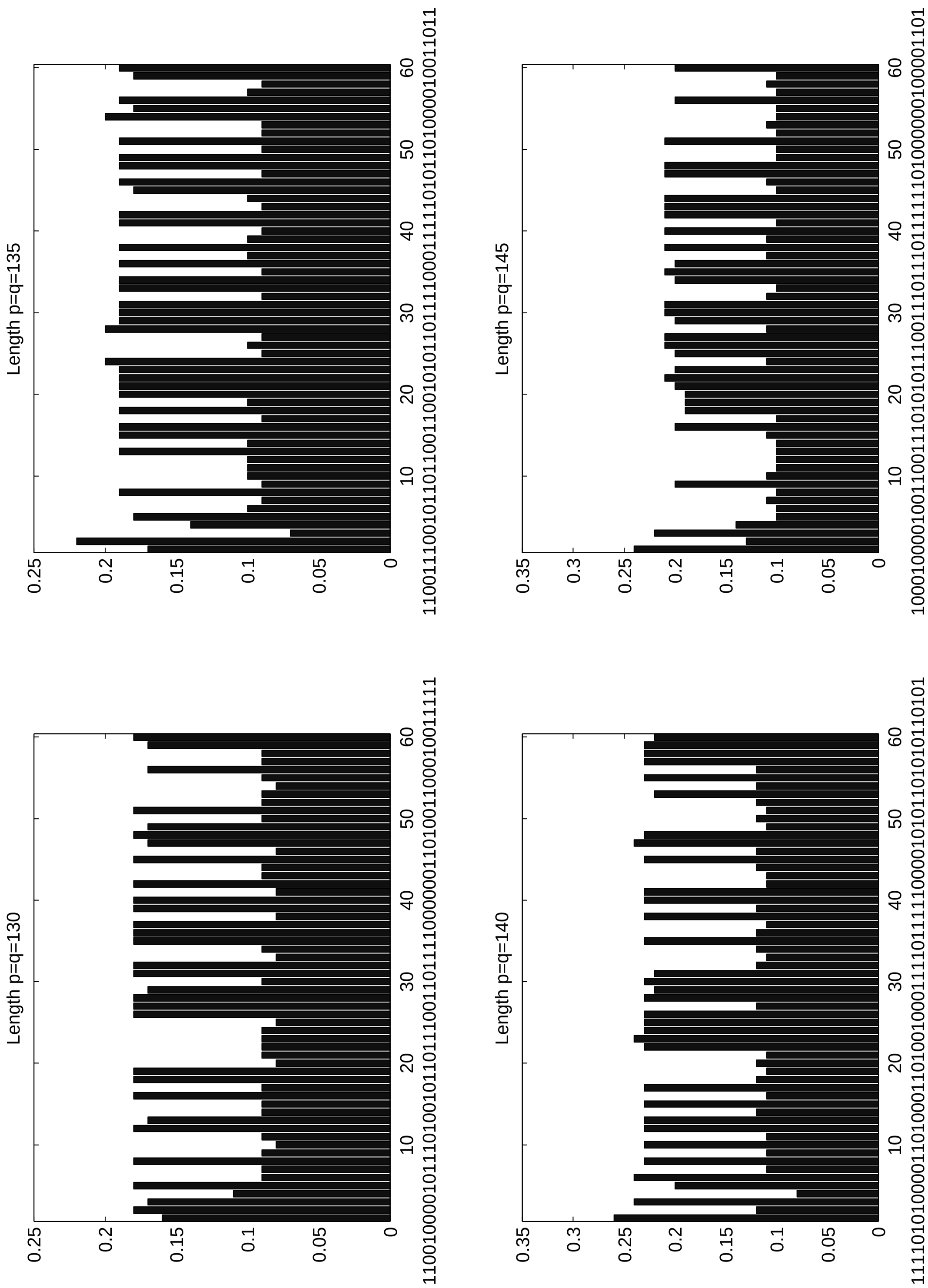


Figure B.14: Timings for an encrypting key of size 60 bit with the size of p and q varying from 130 to 145 bit

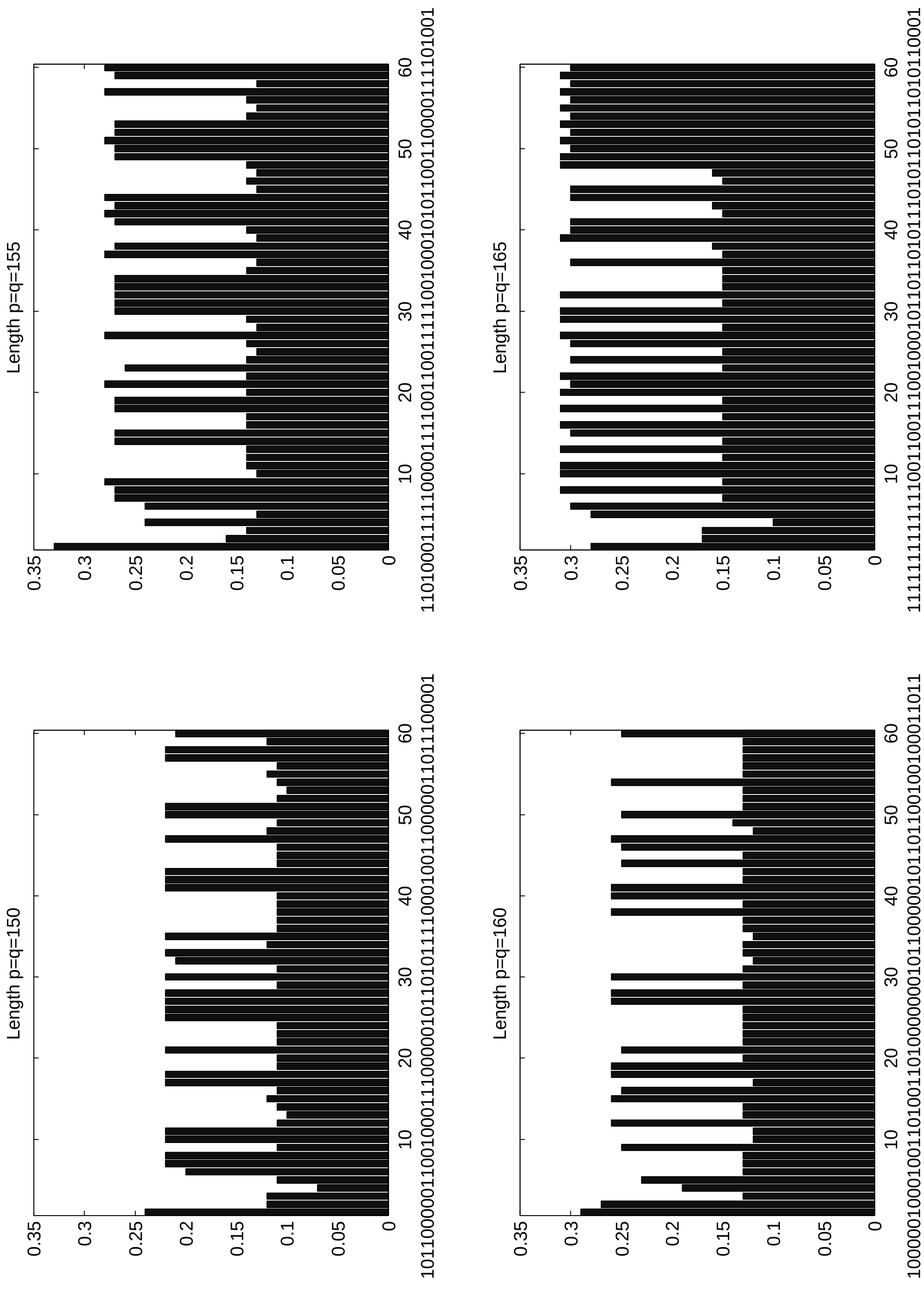


Figure B.15: Timings for an encrypting key of size 60 bit with the size of p and q varying from 150 to 165 bit

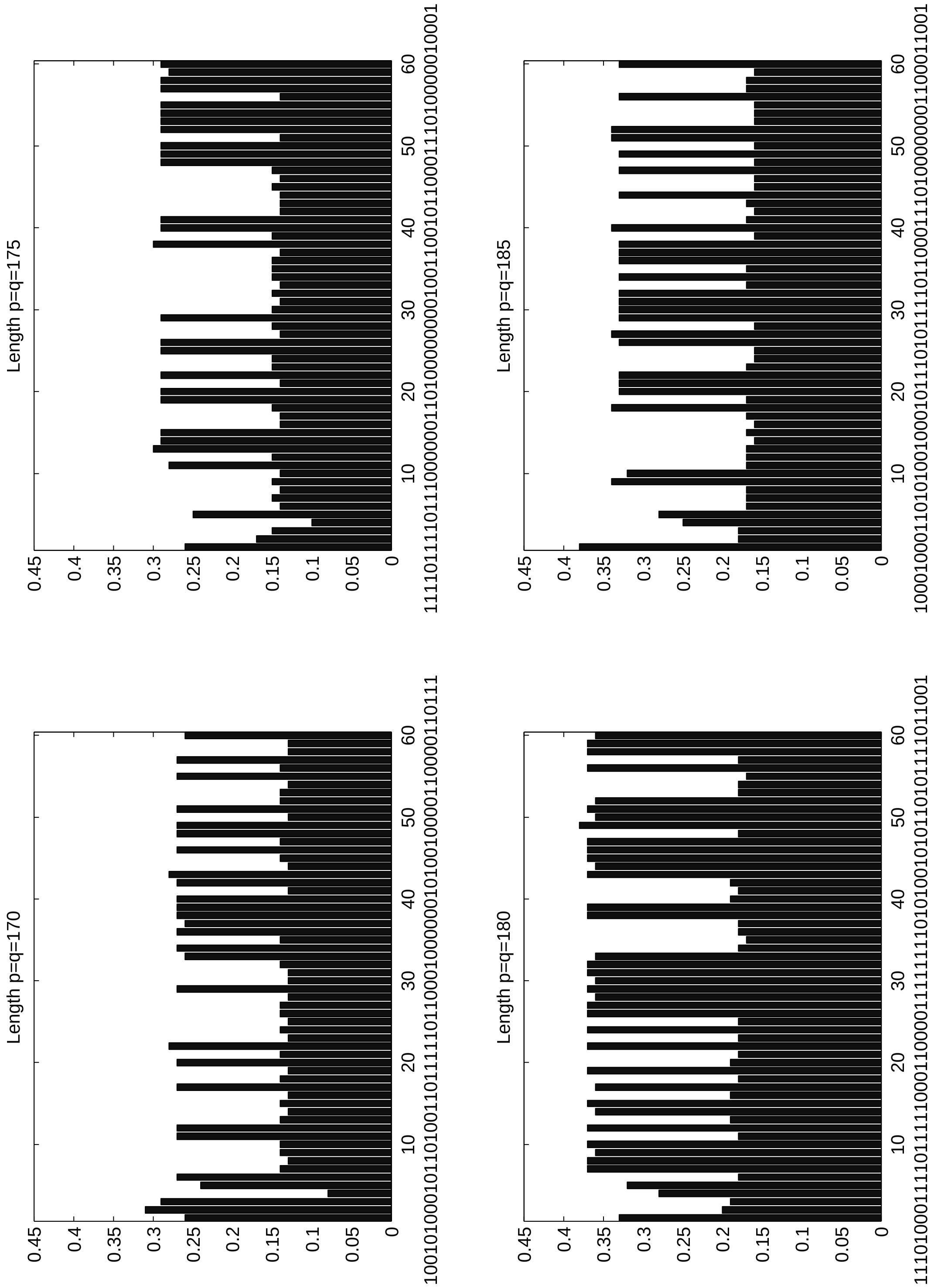


Figure B.16: Timings for an encrypting key of size 60 bit with the size of p and q varying from 170 to 185 bit

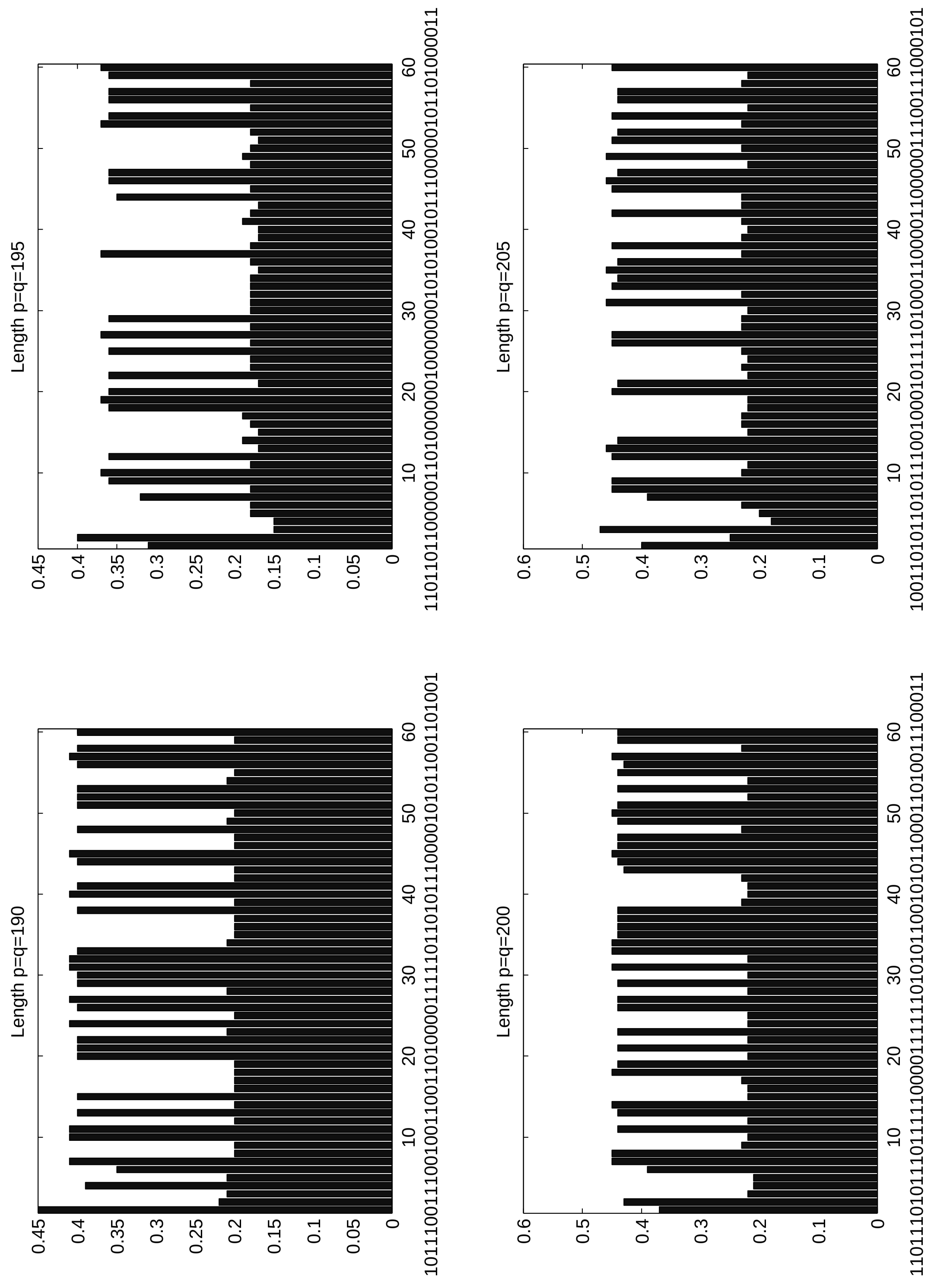


Figure B.17: Timings for an encrypting key of size 60 bit with the size of p and q varying from 190 to 205 bit

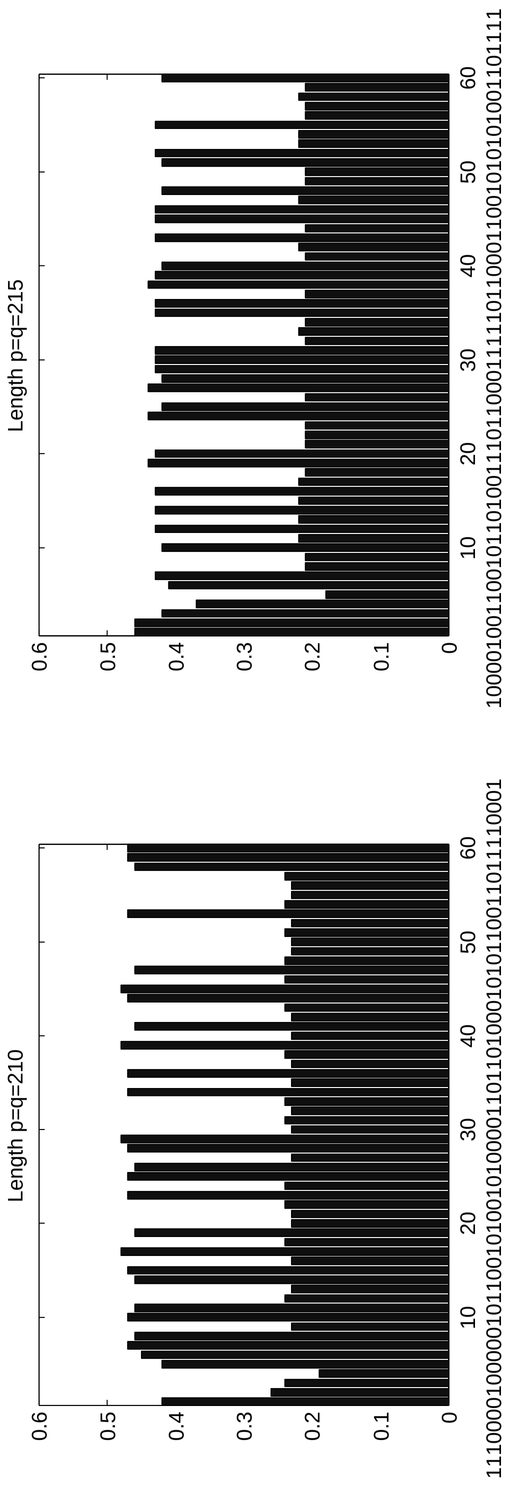


Figure B.18: Timings for an encrypting key of size 60 bit with the size of p and q varying from 210 to 225 bit

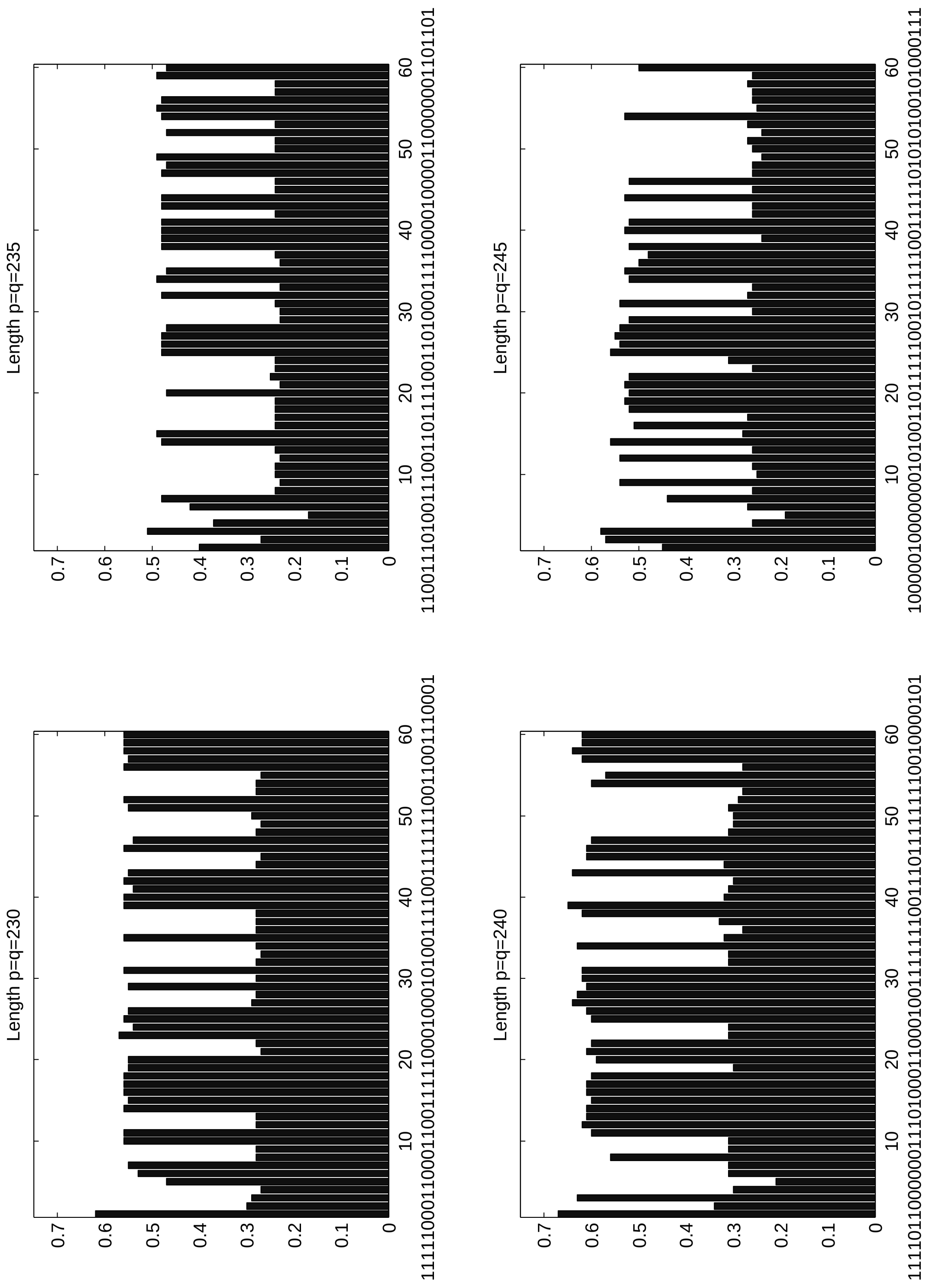


Figure B.19: Timings for an encrypting key of size 60 bit with the size of p and q varying from 230 to 245 bit

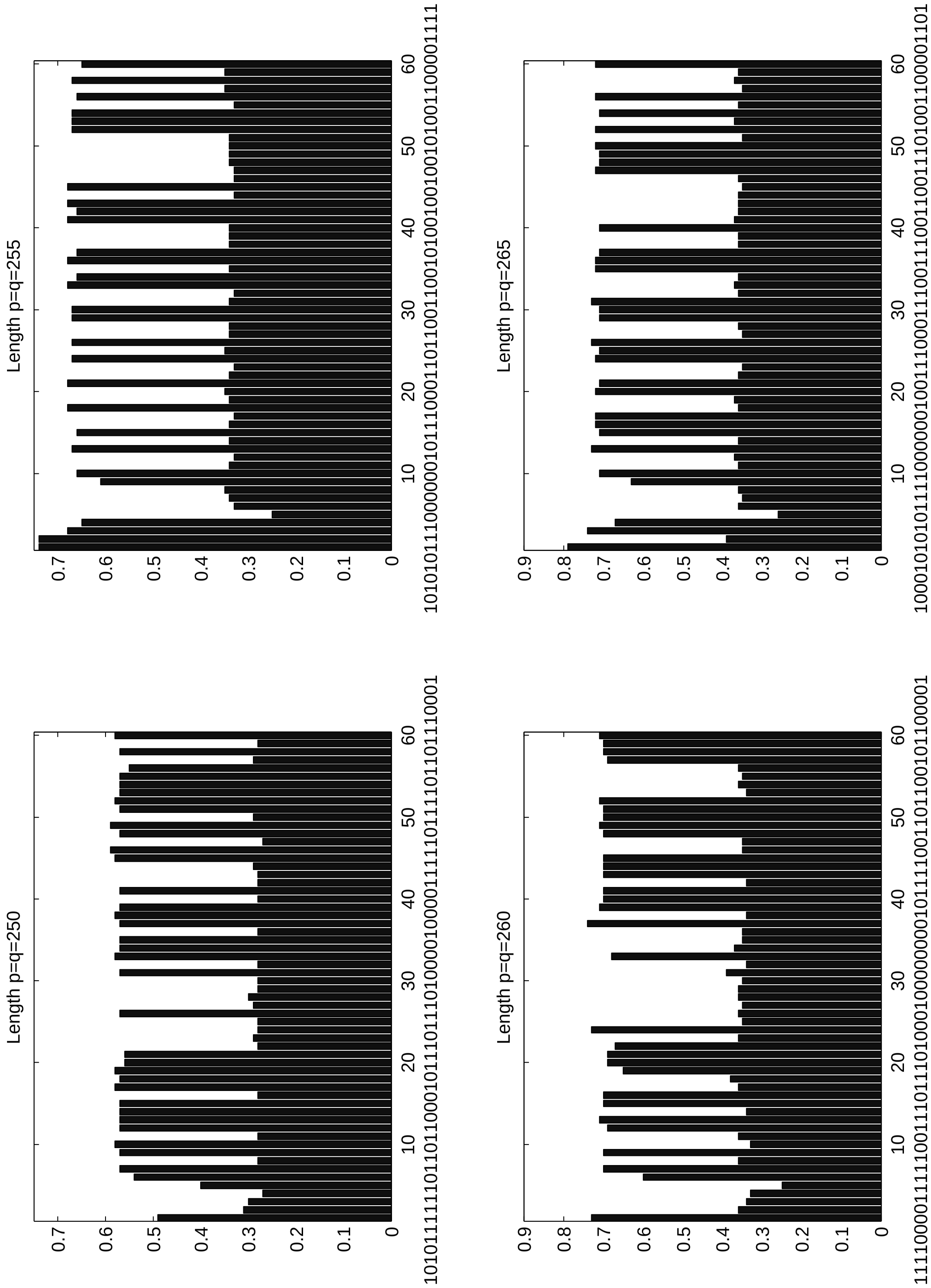


Figure B.20: Timings for an encrypting key of size 60 bit with the size of p and q varying from 250 to 265 bit

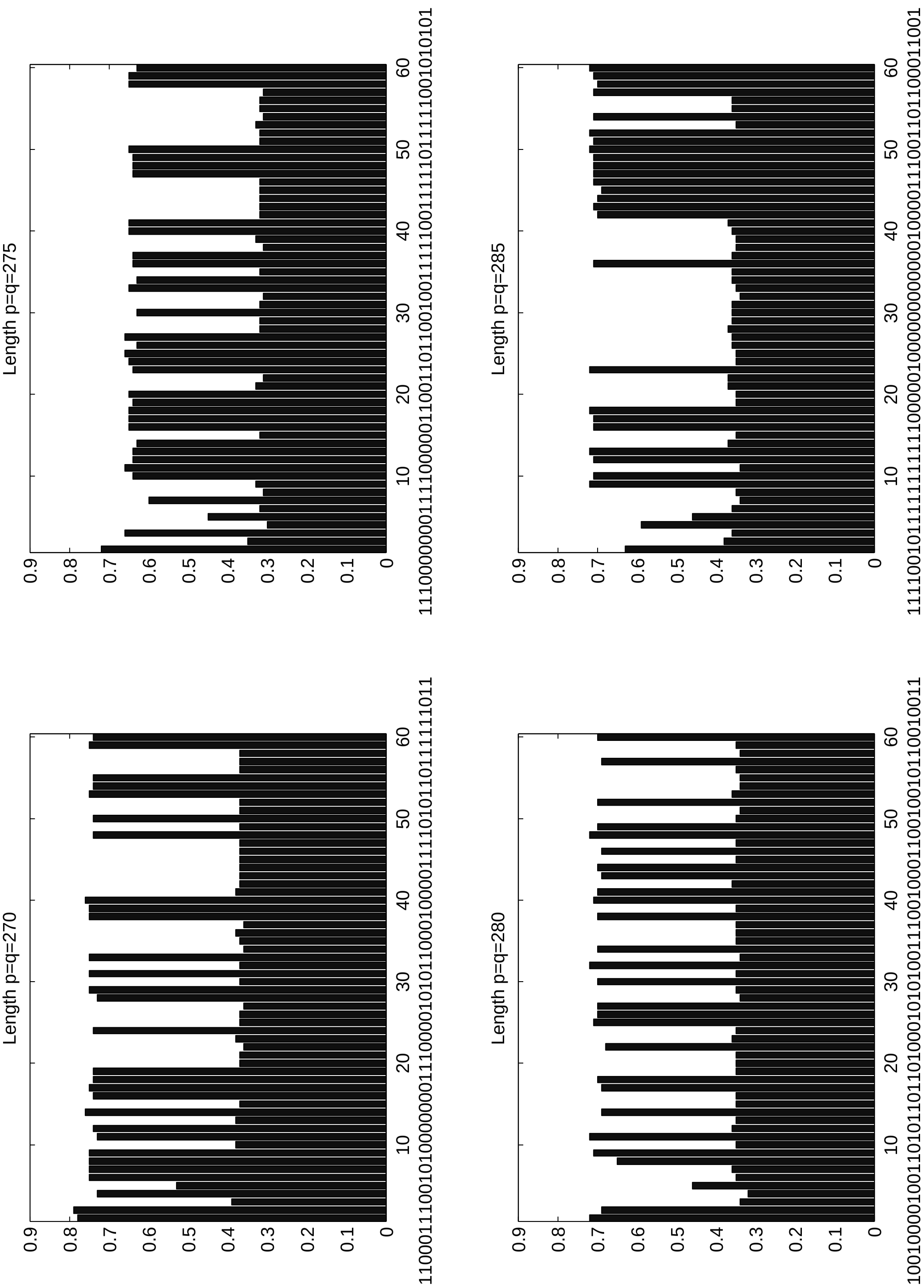


Figure B.21: Timings for an encrypting key of size 60 bit with the size of p and q varying from 270 to 285 bit

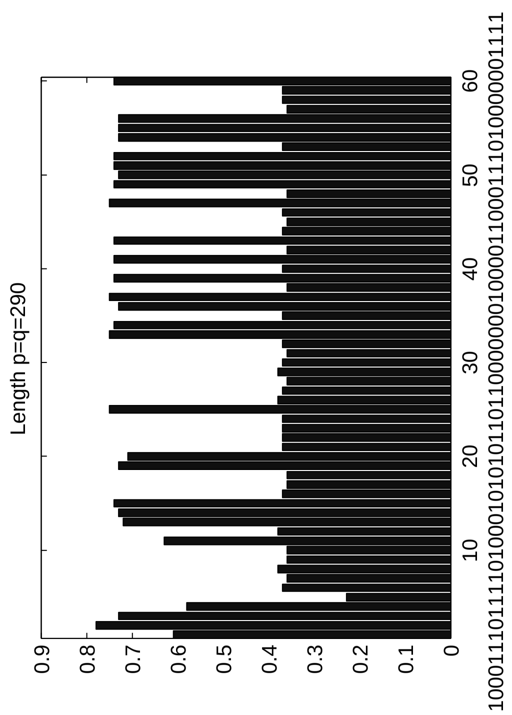


Figure B.22: Timings for an encrypting key of size 60 bit with p and q of size 290 bit

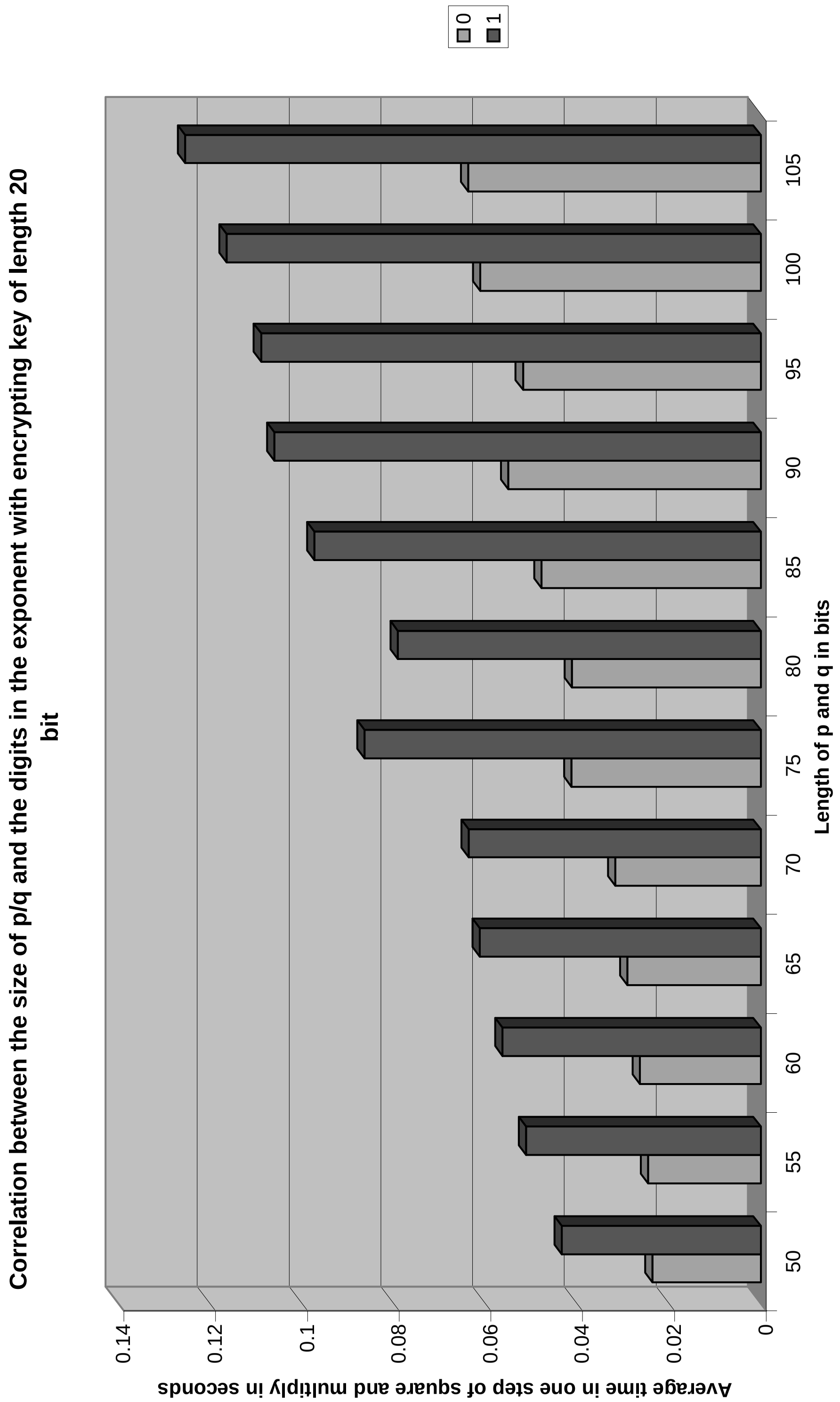


Figure B.23: Correlation between the size of p and q (from 50 to 105 bit) and the digits in the exponent with encrypting key of length 20 bit

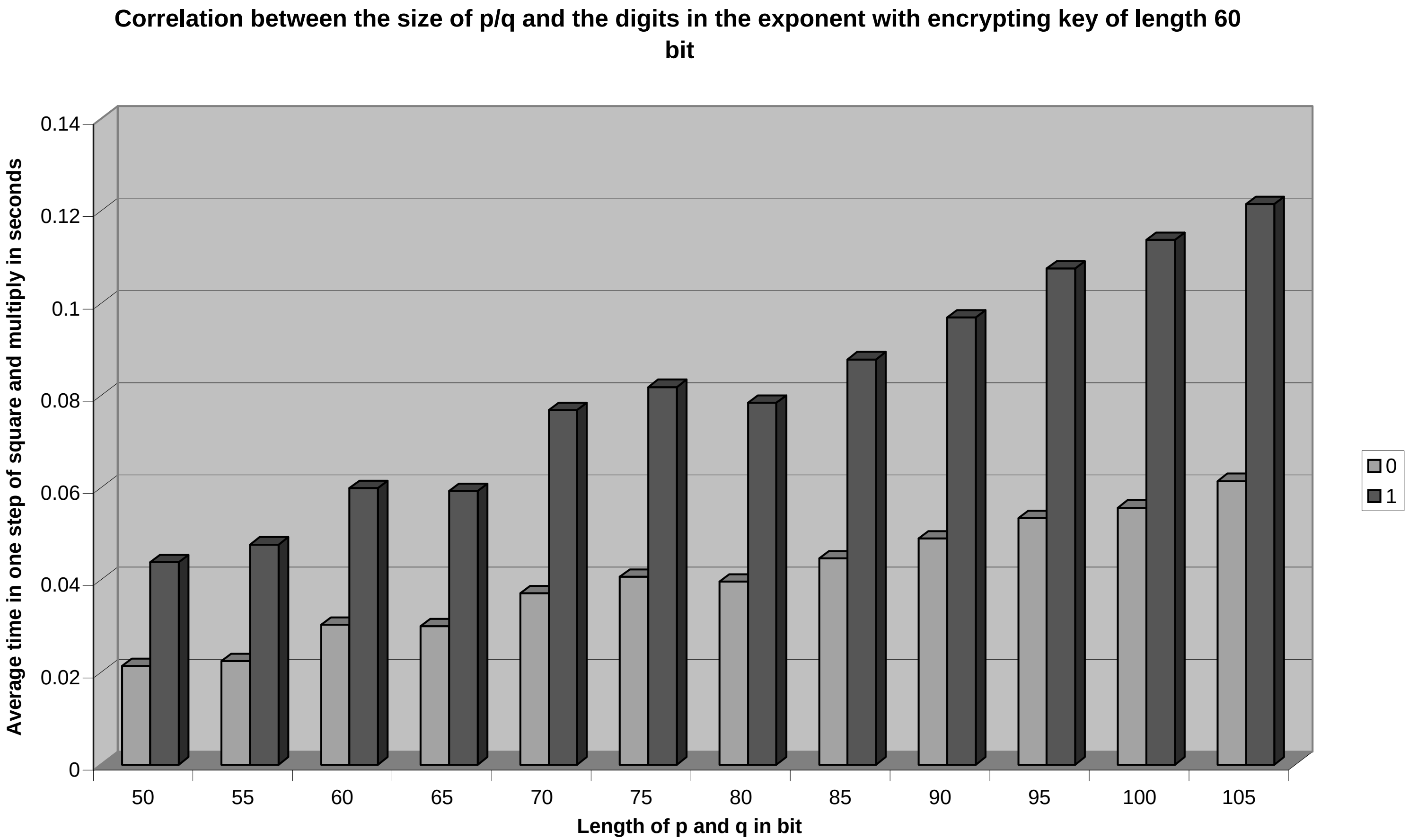


Figure B.24: Correlation between the size of p and q (from 50 to 105 bit) and the digits in the exponent with encrypting key of length 60 bit

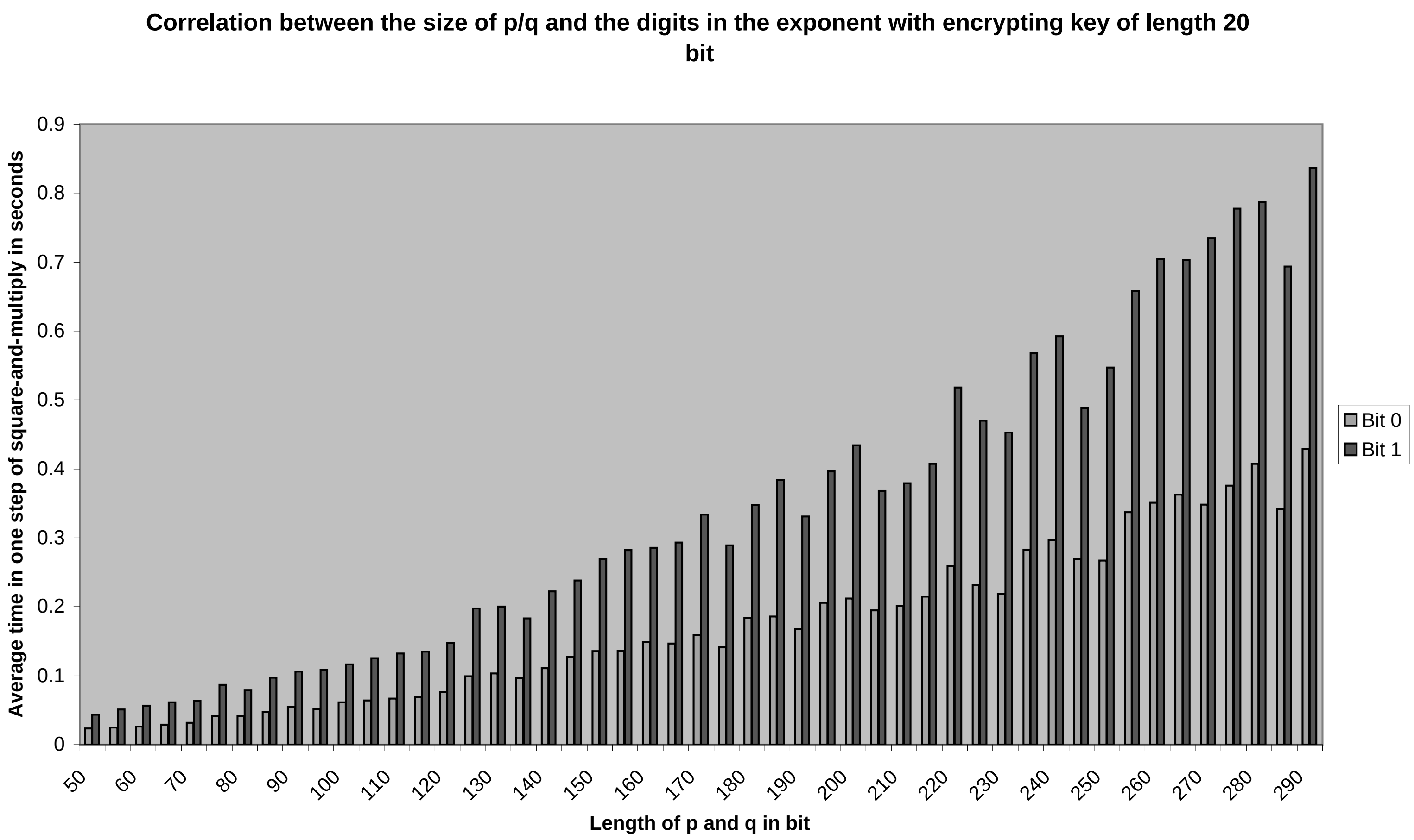


Figure B.25: Correlation between the size of p and q (from 50 to 290 bit) and the digits in the exponent with encrypting key of length 20 bit