RSA: Hacking and Cracking

Ronan Killeen
B.A. (Mod.) Information and Communications Technology
Trinity College, Dublin
Supervisor: Dr. Micheál Mac an Airchinnigh

April 2001

Abstract

It is my objective to investigate the current state of cryptanalytic attacks on RSA, the encryption system invented by Rivest, Shamir and Adleman. We will look at the possible weaknesses in all parts of the cryptosystem including the algorithm, the protocol and the users of the system. I review some factoring algorithms including a description of the Elliptic Curve method of factoring. The spirit of the report is to show that one should not trust to claims that a given algorithm and key length is "unbreakable". RSA has not been proven to be unbreakable, but having survived a great deal of cryptanalytic scrutiny over the last twenty or so years, it has become the de-facto standard and holds a great deal of trust.

Acknowledgements

I would like to thank my supervisor, Dr. Micheál Mac an Airchinnigh, who gave me much of his time throughout the year.

I would like to thank Tim Murphy for his guidance regarding elliptic curves.

I want to thank my Mum and Dad, my brothers, Tomas and Diarmaid, and my sister, Niamh, who have listened patiently to me talk about encryption as if it were the most interesting topic on the planet.

Contents

1 Introduction **1**

2 Background **4**

2.1 Challenges 4

2.2 Early Cryptography 6

2.3 Mechanisation and Electronics 7

2.4 Key Distribution 8

2.5 New Directions in Cryptography 9

2.6 One-Way Functions and Trapdoors 9

2.7 Encryption with RSA 10

3 Student Codes Problem **12**

3.1 RSA E-Test 12

3.2 Preliminary Attacks 13

3.3 Description of the System 14

3.4 Attacking the Problem 14

3.5 Using the power of thought 15

3.6 What about ciphertext only? 15

3.7 Timing Results 16

4 Factorization and Primality Testing **18**

4.1 Why is factoring so difficult? 18

4.2 Prime Numbers 19

4.3 Trial Division 19

4.4 Fermat . . . 20
4.5 Pollard Rho . . . 20
4.5.1 Smooth numbers . . . 21
4.6 Pollard p-1 . . . 21
4.7 Quadratic Sieve . . . 22
4.8 Number Field Sieve . . . 24

5 Elliptic Curves **26**
5.1 What are elliptic curves? . . . 26
5.2 Groups . . . 29
5.3 Pollard p-1 and a General Approach to Factoring . . . 29
5.4 Rational Points on Elliptic curves . . . 30
5.5 Elliptic Curve Factorization . . . 31
5.6 Advantages . . . 32

6 Attacks on RSA **34**
6.1 Exhaustive Search . . . 34
6.2 Low Decrypting Exponent . . . 35
6.3 Common Modulus . . . 37
6.4 Faulty Encryption . . . 38
6.5 Cycle Attack . . . 38
6.6 Low Encrypting Exponent . . . 39
6.7 Adaptive Ciphertext . . . 40

7 Implementation and Results **42**
7.1 Prime Number and Key Generation . . . 42
7.2 Encoding and Encrypting . . . 42
7.3 Knowing a Decrypting Exponent . . . 43
7.4 The Cycle Attack . . . 43
7.5 Low Encrypting Exponent . . . 44
7.6 Low Decrypting Exponent . . . 44
7.7 Other Attacks . . . 45

7.8 Factoring Algorithms 45
7.9 How my implementations fared 46
7.10 A Novel Approach to Factoring 46

8 Conclusions 48
8.1 Lessons to be Learned 48
8.2 If I had more time 49
8.3 Programming with Mathematica 50
8.4 LaTeX 50
8.5 Groups and Elliptic Curves 52
8.6 Researching Skills 52
8.7 Complexity Analysis 52
8.8 Writing 53

A RSA - How it works 62
A.1 Basic Generation 62
A.2 Strong Primes 63

B Tools 65
B.1 Greatest Common Divisor 65
B.1.1 Euclidean Algorithm 65
B.1.2 Extended Euclidean Algorithm 66
B.2 Fast Exponentiation 68
B.3 Primality Testing 69
B.3.1 Pseudoprimes 69
B.3.2 Strong Pseudoprimes 69
B.4 Chinese Remainder Theorem 71

C Source Code 72

Chapter 1

Introduction

> "It may well be doubted whether human ingenuity can construct an enigma of the kind which human ingenuity may not, by proper application, resolve." -Edgar Allan Poe, *The Gold Bug*.

Everyday the RSA cryptosystem is used by millions of people (whether they know it directly or not). From credit card transactions over the Internet to communications between banks, to emails encrypted with the Pretty Good Privacy package, RSA has become the "de facto" standard for public key cryptosystems. But the security of RSA has never been formally proven. The trust placed in it is based upon over twenty years of concerted attacks from expert sources. Can we continue to trust a system that is twenty years old, to play such an important role in our transactions?

One of the best ways to prove that a given system is secure, is by trying to break into it. This gives first hand experience of the possible vulnerabilities of the system. Dan Farmer and Wietse Venema suggested [11] that site administrators should do just that - attack their own organisation's Unix system. They argue that the most dangerous sort of attackers are "uebercrackers" - individuals who have expert knowledge of systems. Their knowledge would be on par to that of the administrators themselves (if not better). Uebercrackers also have the ability to write and modify hacking programs to exploit weaknesses of services provided by the system. When the people in charge of security see the system from the attacker's point of view, they get an appreciation for the security of the system. Only then can informed decisions be made about security policies.

In the original RSA paper [19] "the reader is urged" to find a method to break the system. If the cryptosystem was to survive concerted cryptanalytic attack for a long enough period of time then we can accept its security to some extent. This is in fact what has happened with RSA. In the twenty years and more it has withstood huge pressure and still remains the most widely used public key system.

Shortly after the M.I.T. cryptosystem (as it was known at first because it was invented there) was announced, Martin Gardner published a challenge in his Mathematical Games article [12]. This challenge gave a public key and a short message encoded with the key. The person (or people) who cracked the message were to win a $100 prize. It was not until 1994 that the challenge was completed by a team who distributed the workload over the internet. Although the key size was not as large as those used in the industry, since then challenges of increasing difficulty have been attempted and met. Most recently a 512-bit key (RSA-155) was factored using the General Number Field Sieve algorithm.

In chapter 2 we will investigate the background to RSA and to cryptanalytic attacks. Chapter 2 describes my breaking of Micheál's student code system. Chapter 4 examines the different factorization techniques that are available, and considers their strengths and weaknesses. Chapter 5 takes a more detailed look at the Elliptic Curve method of factoring numbers. In Chapter 6 some of the attacks that do not depend on factoring the public modulus are discussed. Chapter 7 describes some of the implementation difficulties and results. Finally Chapter 8 describes the results of my attacks on RSA and concludes the report.

There are also several appendices with additional information. Appendix A gives a brief description of RSA key generation. Appendix B provides a description of some of the algorithms that are necessary for RSA and also for some of the attacks.

Please note that there is a CD-ROM with material that should be distributed with this report. It contains the Mathematica notebooks used in the project (and listed in Appendix C), the web-page that I wrote during the course of the project, a selection of interesting papers that I obtained electronic versions of, and several other interesting resources. The CD-ROM is optimised for Windows systems, but should work on most major operating systems. Please note that the URL for the web-site is:

`http:\\members.tripod.com\irish_ronan\rsa\`

Chapter 2

Background

2.1 Challenges

In the his column *Mathematical Games* [12], Martin Gardner published a challenge on RSA. His article gave a brief description of RSA, the challenge key and demonstrated the encryption algorithm. There was also a signature, encrypted with the private key to prove that the system worked without revealing the private key. The decoded signature read:

FIRST SOLVER WINS ONE HUNDRED DOLLARS

The interest generated was enormous. In April 1994 a group of about 600 volunteers successfully factored the modulus. They used the distributed power of the Internet to co-ordinate the attempt that took eight months. Each member of the group used their own computers' idle time to perform the necessary calculations. The algorithm used was the Multiple Polynomial Quadratic Sieve (MPQS) and is described in Section 4.7. A report on the task by Atkins et al [3] was named "The Magic Words are Squeamish Ossifrage", which was the decrypted message.

Public challenges are a good way to keep track of the state of the art in a given field. Fermat used to issue challenges to contemporary English mathematicians. His reasons may have been more to infuriate them and to show off his superior intellect than to improve things. After his death, many of his theorems, written in the margins of his copy of Diophantus's *Arithmetica* were discovered, but few were proven formally. Fermat had a habit of not completing his proofs, yet one by one in the years after his death, other mathematicians proved these theorems. However, his Last Theorem (that $x^n + y^n = z^n$ has no whole number solutions for n greater than 2) would prove one of the most well known and difficult problems in mathematics. It is only recently that Andrew Wiles found a proof. By doing so he made huge advances in number theory including some work on elliptic curves. An excellent book about the circumstances surrounding this theorem is [24].

RSA Security Inc. released additional challenges in the early '90s. The challenges were of increasing difficulty, starting with a 100 decimal digit key. Awrads (all greater than the $100 of the previous challenge) were awarded for success. The latest of these to be broken was RSA-155, which contained 155 decimal digits (512 binary digits). The effort took four months and is described in Lenstra et al [6].

Another more general set of numbers was also released. These are called the Partition Challenge numbers. Unlike the RSA numbers which have only two large prime factors, the Partition numbers may contain many and smaller factors. The partition number p(n) is the number of ways of expressing n as a sum of positive parts. For example, p(5)=7 as

$$\{5,\ 4+1,\ 3+2,\ 3+1+1,\ 2+2+1,\ 2+1+1+1,\ 1+1+1+1+1\}$$

The challenge list contains the partition numbers of all the primes from 8681 to 20000 (they range from 100 to 170 digits long). The entire list can be generated in 30 minutes on a Sun SPARC workstation. The algorithm and more information on partitions can be found in Andrews [2]. Many of the partition numbers are easy to factorise, but there are some that are extremely difficult to factor. Both challenge lists and some Common Lisp code for the generation of these numbers are available on the CR-ROM accompanying this paper.

2.2 Early Cryptography

My first introduction to cryptology was in primary school. We were shown the Caesar cipher. You take each letter of the secret message and replaces it by the letter that follows three places after it in the alphabet. This leaves the ciphertext looking like rubbish to the uninitiated eye. Using this scheme (possibly with a different offset) my friends and I were able to communicate without anybody knowing what schemes we were up to. However knowing what cipher was being used was enough to be able to break it by trying several different offsets.

Later it became clear that secrecy alone would not protect a cipher. When the details of the systems became known, cryptologists developed the idea of an expanded key-space. The value of the key would determine the ciphertext and not just the method. For example, the Caesar cipher described above used a key of 3. That means that the letter should be shifted three places. Any number could be used from 1 to 25 in the English language. It would not take long to check the entire key-space to find the correct key.

To expand the key-space we could replace the letters of the message by a random letter. The key ends up being a map of the letters onto their pairs. In fact new symbols can be invented to obscure the message further. For the English language, with 26 letters there are 26! different combinations of keys. This figure is about 4×10^{26}. Of course there are far too many possibilities to check, which makes this seem like a reasonable system. However, a simple technique can be used to crack the system, given a long enough ciphertext. We perform a frequency analysis on the text. That means we count the number of times each symbol appears. If we know the language of the text we should know the frequency that certain letters appear in that language. For example, in the English language, the letter 'e' is the most common, followed by 't' and then by 'n'. This means we get clues to the plaintext. By using frequency analysis on digrams (pairs of letters) we can gain further information. For an excellent tale related to cracking a mono-alphabetic cipher of this type, read Edgar Alan Poe's *The Gold Bug*.

Various other schemes were devised and one by one cracked through ingenious means. Through literature such as Poe's tale, the public soon became interested in cryptography. Forbidden lovers would print enciphered messages in the newspapers to express their love. Military leaders still used encryption for more serious affairs, but these systems were being broken regularly by the so called "Black Chambers". These dark and secret rooms were often in the same building as the city's post office. Teams of workers opened, copied, and resealed messages to avoid detection. Later the cryptanalysts would attempt to decipher the messages. The Black Chamber of Vienna was very successful in breaking encryption and sometimes sold the information for a high price.

2.3 Mechanisation and Electronics

The cryptographers gained an advantage over cryptanalysts with the mechanisation of the field. The Enigma machine was used by the Germans during World War II. The system used three disks, which each did a substitution. An electric signal, initiated at a keyboard, passed through the three disks. A reflection board then turned the signal back along another pathway. The key to the scheme was that after each letter was encoded the first wheel would rotate. When it had rotated fully the next wheel would move one place. Finally by the time the second had rotated completely, the third start to move. Therefore each letter would be encrypted differently each time it was used during the sequence. Also the system could be started in many different positions. Each day a new starting position would be used.

But this system was broken using some incredible mathematics. A Polish mathematician, Marian Rejewski, noticed a flaw in the system. He developed machinery of his own that was able to capitalise on this weakness. However he did not have the resources when the war broke, as the Germans attempted to strengthen their cipher. Bletchley Park in England continued his work and by the middle of the war were routinely able to break Enigma. They capitalised on messages that they could guess the plaintext of, e.g. the weather report at 6 a.m. always began with "wetter". This is what is called a known-plaintext attack. This head start cut down dramatically the amount of work their machinery had to do to find that all important starting position of the disks. Having done this they could then decrypt every message that followed that day. This was a huge tactical advantage for the British.

The advantage gained when computers began to be used was immense. The increased speed and complexity allows extremely secure ciphers to be built. Systems like DES (Data Encryption System) allow information to be transfered between two people with a shared key. However, DES uses a 64-bit key that is effectively only 56-bits (each block contains a parity bit which is redundant). Therefore it is not impossible to use an exhaustive search for the key. Distributing the test over a network is useful, as the idle time of workstations can be used to plough through the possibilities. There are 2^{56} key possibilities but this is not enough.

RSA Security sponsored several DES cracking competitions to highlight the need for stronger encryptions. The latest, DES-III, was cracked in a record 22 hours using the combined computers of distributed.net and their members. Over 1000 users used their personal computers to search a small part of the key space, checking an incredible 245 billion keys per second. Even when stronger key lengths are used, we still have the problem that they may be broken eventually.

2.4 Key Distribution

The 56-bit key of DES is certainly not secure against an attack distributed over the Internet. But if an attacker had access to a large number of computers (a network in a university for example) then they will be able to mount an attack that would take a reasonable amount of time. During the '70s very few had access to the resources to mount such a search, but many people were worried about the length. The general consensus was that a key would remain secure for a limited time only.

The result of this is that new keys must be created regularly. The key distribution problem is about getting the new keys to the person to whom you wish to communicate. As businesses and banks expanded and needed to communicate globally in a secure fashion, this became extremely difficult to do. The only safe way is to give the key personally to the second party - transmitting it would be dangerous, even with a special master key. The reality was that couriers would travel to the other party with the key in a briefcase handcuffed to the couriers arm. However this becomes extremely expensive, especially if there are many organisations with which you need to communicate.

2.5 New Directions in Cryptography

In 1976, a ground breaking article was published by Diffie and Hellman [8]. It outlined the requirements for a public key system. In simple terms, the idea was as follows: One publishes the means by which to encrypt a message and keeps the means to decrypt it secret. It should be extremely difficult (i.e. time consuming) for an attacker to construct the decrypting method themselves given only the encrypting function. To transfer a message, check their public algorithm, use it to encrypt the message, and then send it. They will be able to decrypt it but no one else will. It doesn't matter who sees the algorithm, because they should not be able to do anything with it. The public information is however the best target - no unorthodox means is need to be used to get it, like intercepting an email transmitted over the internet.

2.6 One-Way Functions and Trapdoors

An encryption system like DES uses a two-way function. If you can encrypt the message using a key, it is easy to reverse the process. When two people share a key, they can both encrypt and decrypt messages encrypted with those keys.

We introduce the idea of a one-way function. We can do something easily in one direction, but it takes too much effort to reverse the process. As a simple example, consider multiplying two large prime numbers together. It is easily done with a pen and paper. But to reverse the process - to find its factors is incredibly difficult. This is an example of a one-way function.

In cryptography, if we could publish a one-way function, and tell people to use it to encrypt a message (encoded as a numerical sequence), then nobody would be able to reverse the process. In fact it does not seem to be useful, because we would not be able to read the message either. But what if there was some little bit of secret information that we did not publish. This piece of information allows us to reverse the process. It is a trapdoor to the system. Now an associate could use the algorithm to send personal data. Nobody else would be able to read the data in its encrypted form. But when we receive it we could use the extra bit of information to decrypt the message. RSA uses a one way function that has a trapdoor.

2.7 Encryption with RSA

An RSA public key contains two numbers: an encrypting exponent, e, and a public modulus, n. To encrypt a message, we first encode it into numerical form, in blocks that are less than n. We then perform the following function on each of the blocks, b_i,

$$c_i = b_i^e \ (mod\ n)$$

where c_i is the corresponding ciphertext block. This is called fast exponentiation, and can be done reasonably quickly in both software and hardware (see Appendix B.2).

The decryption function is of exactly the same form. This time, however, we use the private key, which consists of a decrypting exponent, d, and the same public modulus, n. To retrieve the blocks we do the following calculations:

$$b_i = c_i^d \ (mod\ n)$$

Signing a message uses the same idea. This time the plaintext blocks of the message are encrypted with the private key. The nature of the system means that the public key will decrypt this signature to the original plaintext. The sender sends both plaintext and the signature to a recipient, who then checks to see that the plaintext matches that recovered from the signature. Since only the private key could have done the encryption, and only the sender has access to it, the message must have come from that person.

In practice it is not necessary to encrypt each block to make a signature. The sender should use a hashing function which reduces the message to a single block. Given this single block it is impossible to rebuild the message. Given the message and the hashed block, however, it is easy to verify the hash. The sender then signs the hashed block. On receiving the plaintext and the signature block, the recipient can rehash the plaintext, and check to see if it matches the signature that is decrypted with the sender's public key.

The details of generating an RSA key pair are in Appendix A. In short, the public modulus, n, is chosen by first selecting two very large prime numbers and multiplying them together. The encrypting exponent, e, is chosen carefully so that it has an inverse. If the factorisation of n is known, then one can easily compute d, the decrypting exponent. However, this information is the trapdoor information and is kept secret. Therefore, to break the system, one needs to be able to factor n. At least, no one has discovered a faster attack for large keys.

Chapter 3

Student Codes Problem

3.1 RSA E-Test

Micheál runs a electronic test with his first year students which involves a question on RSA. He generates a public and private key pair for every student and then requires them to decrypt a message. Each student gets a different message. Originally Micheál used a scheme whereby the public exponent was derived from the student's unique identifier that is used for college exams. But it was pointed out to him that it was against college regulations to publish student names alongside recognisable student numbers. This is because the student numbers are used to publish exam results and these are considered private.

To solve the problem, Micheál decided to use a simple scheme for encoding the numbers so that they are no longer recognisable to the casual observer. Unless one knows the algorithm and the parameters (the key) one cannot derive the student numbers from the codes, which are printed beside student names. Micheál challenged me to try to break this code as a first task before attacking RSA directly.

3.2 Preliminary Attacks

Before starting I was given some hints. Micheál told me that it involved a multiplication modulus some number. He also told me that the process was completely invertible. This was an excellent place to start and a big hint to start experimenting. I began by trying to invent a similar scheme myself to try to get an idea of the best way to attack the problem.

The hints suggest a multiplication of the following kind:

$$code \ = \ id \ \times \ multiplier \ (mod \, m)$$

where m is some large number. The inverse of this can be found using the extended Euclidean algorithm (see Appendix B.1.2).

I started by choosing a large prime number as the modulus. I noticed that none of the codes had more than 8 digits. This led me to believe that the modulus should ensure this. I chose the first prime above 8 digits . This is the prime number 100,000,007. Then I tried various multipliers and applied them to the and found that to get an even spread of codes, one could use most numbers above several hundred.

I then found the inverse multiplier. This is a number that can be multiplied by the code modulus the same m to reveal the original id. I was interested to see if there was a difference in the order of the multiplier compared to its inverse. I found that there was a considerable difference. Even if a small multiplier is used, its inverse is probably much larger. This would mean that it would be harder to find the id from the code because the inverse search space is larger. However if the multiplier is chosen well, both it and its inverse should be large. This looks like a daunting task - I know neither the modulus or the multiplier and the search space is huge.

What follows is a more detailed description of the problem that benefits from hindsight. Then I present my approach to the problem and some thoughts about the difficulty of the task.

3.3 Description of the System

Micheál has a list of students and their student identifiers. He wishes to create for each student, a unique student code derived from the id. He wants this to be secure, that is, no one should be able to derive the id from the code. The code is to be published on the web alongside all the student names as part of an electronic-test.

Micheál chooses a modulus, m, that is coprime to all ids and is larger than each. He then chooses a multiplier, n. He next performs the following function:

$$c_i = s_i \times n (mod\ m)$$

where c_i is the corresponding code for student id s_i. Once the ids have been converted, the next step is to add several digits randomly to each number to make them into public exponents for the RSA system. Micheál used a Perl script to do this.

Inverting the codes back to the ids is a similar process. We first of all have to find the inverse multiplier, n^{-1}, and then perform the following function:

$$s_i = c_i \times n^{-1} (mod\ m)$$

which returns the student id. To find the inverse we use the Extended Euclidean Algorithm (for finding the greatest common divisor) to derive the equation:

$$(l \times n) + (k \times m) = 1$$

where $l, k \in \mathbf{Z}$. From this we can see that l is the inverse of n. For more details about the algorithm please see Appendix B.1.2.

3.4 Attacking the Problem

Starting with the next prime following the id, try a given range of multipliers, and then go on to the next prime. When I say "try", I mean multiply the id by that multiplier modulus the prime, and see if it matches the corresponding code. I decided on quite a large range of multipliers (500 to 9000) and took the primes in sections. I would first generate a list of primes and then apply the test to each.

Having run the test on some test data that I generated to make sure the program ran according to plan, I started the test. I realised that the test could run for quite a long time so I included a simple printout mechanism that allowed me to see how far the algorithm had progressed. This would allow me to stop the test and restart it at the point it left off, if I ran out of time.

Fortunately, after leaving the computer running for about a couple of hours (through a long lunch), I got a result. I have to admit, I was expecting to have to try a much wider range of multipliers before I got a result. The modulus turned out to be 99842251, and the multiplier was 6794.

3.5 Using the power of thought

I had used the computing power that was available to me. But was there a better way to go about things. What was so special about the modulus? As it turned out, to get the modulus Micheál calculated the maximum student id and found the next prime after that. I knew already that the modulus had to be bigger than all of the ids for the system to work, yet I chosen to start testing with primes that were only higher than the id I chose to test with. Had I thought about it more clearly, instead of jumping straight in I would have had success sooner. Luckily Micheál did not choose a large value for the multiplier. Had he done so my testing would have had to go on much longer.

3.6 What about ciphertext only?

Before I started the attempt, I considered using only the student code to crack the system. This is a much more difficult task for a number of reasons.

Firstly, I would have no way to guess at the modulus, having no access to the highest student id. All I would know would be the approximate range of ids. I would then guess at a modulus, starting at the prime 100000007 and dropping down. I would have to do the calculation on code, hoping to get the right inverse multiplier. If it landed within the range of expected ids, I would test other codes with that inverse multiplier to see if they dropped into the correct range. But even if they all do - can I be sure that these are the correct ids.

Let's make the situation less complicated. Assume that I can get my hands on the student ids for the class, but do not know which id belongs to what student or what code. I now have a slightly simplified task. Using my experience I might guess the correct modulus, based on the list of student numbers. Then my task is to try all possible inverse multipliers to see which ones convert the codes into a number in the id list. It is not unreasonable to assume that I could retrieve a list of ids, especially since I have access to many resources - notice boards, course web pages etc. Even a partial list would be enough.

3.7 Timing Results

In my rush to get the test underway I forgot to take advantage of Mathematica's timing capabilities. The simple "Timing" command can be used to check how long a program runs for. Instead of running the test in its entirety again, I decided to time the length for a single block of 500 primes to work. From noting how many blocks were tested by the program, I was able to give a guess to the runtime.

No. of Blocks of 500 primes	Time to Check (sec.)
1	223
25	5575

Table 3.1: Timing Extrapolation

From the table it shows that the 25 blocks took 5,575 seconds which is roughly an hour and a half.

Chapter 4

Factorization and Primality Testing

The most obvious method of attack on RSA is to attack try to factorise the key itself. If successful, one can then read any message sent encrypted by this key. We will look at the development of techniques. The Elliptic Curves method however gets a chapter on its own.

4.1 Why is factoring so difficult?

We can multiply two numbers together very easily and quickly, but when it comes to factoring them, that is figuring out what numbers made them up, we are faced with a problem of much greater complexity. Look at the number 283853 which has two prime non-trivial factors. Judging from the last digit we can gather that if one of the factors ended in a 3, the other would have to end in 1. However if one of the factors ended in 9, the other would end in 7. We might try to continue with the second digit having chosen one of the two possibilities to test first. However, there are too many possibilities for the second and further digits.

The problem is that there are very few clues given by the number to its possible factors. The number of possibilities grows exponentially. Before beginning this project I had tried to write a program that generated the factors based on the technique in the previous paragraph. My algorithm worked with the binary representation of the numbers. For each bit of n, there are two possible pairs of factors. My algorithm would make guesses at each bit backtracking whenever it became clear that the number was wrong. This algorithm has a terrible running time $O(2^n)$ where n signifies the number of bits in the number to be factored. This is an exponential time algorithm and is very slow. The code will be included on the CD-ROM accompanying this report.

4.2 Prime Numbers

Primality testing is important for the RSA system. Our public key depends on being able to choose large random primes. If the algorithm to compute and check that the numbers generated are actually prime is faulty, then it endangers the entire system. With RSA it is likely that any error will easily be noticed, however, when attempting to encrypt. However it is important to know the risks.

Primality testing is linked closely with factorising. I have described the pseudoprime and strong pseudoprime tests in Appendix B.3. But we will see how certain factorization techniques such as Trial Division and Fermat are also (bad) primality tests.

4.3 Trial Division

A simpler method to factor a number is to simply try to divide numbers less than the square root of the number. If a number divides evenly with no remainder, then you get a factor (and also its couple - the number you multiply by it to get the original). This can be improved by not dividing by even numbers if the original is odd, because an even number multiplied by any other always gives an even number. There are other ways to speed this process up, such as trying only prime numbers, but even still, we will have to perform a large number of calculations, making this form of factorization very slow for numbers with greater than 11 or 12 digits. The running time for trial division is exponential.

4.4 Fermat

Fermat's algorithm works from the square root of the number, n, and works outwards. It tries to find two numbers, x and y, so that $x^2 - y^2 = n$. This is the difference of two squares and can be written $n = (x - y)(x + y)$, and thus n is represented as the product of two integers.

Although the algorithm can be sped up by keeping track of certain variables, this particular algorithm is not suitable for factoring large numbers. The problem with both trial division and Fermat's algorithm is that they do not only find factors, but if left running long enough, they prove primality. They are deterministic and will check every possible factor before returning. This is a particularly bad way of proving primality, especially since we have the pseudoprime test. The rest of the algorithms described will not be deterministic but probabilistic.

4.5 Pollard Rho

Pollard Rho and Pollard p-1 factoring algorithms go about factoring in a new way. Before, with trial division and Fermat, if we left the algorithm running for long enough, every factor would be found. If there were no factors then we would have proved primality. The rest of the algorithms that I will describe are probabilistic tests. They search a portion of the search space, starting from a random point and run for a given amount of time. If no factors have been found, then they can be started at a new point. The advantage of this is that we have a better chance of finding factors faster.

The Pollard Rho algorithm is initialised with an irreducible polynomial, f(x) and an initial value for x_0. We generate $x_i = f(x_{i-1})\ (mod\ n)$ for $i > 0$. The y values are defined as

$$y_i \ = \ x_i\ (mod\ d)$$

where d is a non-trivial divisor of n. Eventually we will get to a pair of cases where $y_i \ = y_j$ (as there are only a finite number of congruence classes modulo d). When this happens, we will know that it will continue for every following pair $y_{i+t} = y_{j+t}$. When we get to this state, we know that $x_i \equiv x_j\ (mod\ d)$. Whenever $x_i \neq x_j$ we can get a factor of n from gcd(n,$x_i - x_j$).

Of course we do not know d, the divisor, in advance. This means we cannot know any y_i either. So we will never know when we get to that stage where the ys run in a repeating loop. The only way to do this is to systematically choose pairs (i, j) and running the test gcd(n, x_i, x_j). It is usually a good idea to multiply several of these together and get the gcd of a combined number, to save on the algorithms running time.

4.5.1 Smooth numbers

Before discussing Pollard $p - 1$ and the Quadratic Sieve we must define smoothness. A number is said to be b-smooth if each of its prime factors is less than the bound b. When we say that a number is b-smooth for some bound, we are really saying that we have the ability to completely factorise it. The bounds we use are usually within the range of more ordinary factoring methods.

4.6 Pollard p-1

The name comes from the fact that $p-1$ contains small factors (where p is a divisor of n), then this algorithm will work quickly. To show this I will state a theorem that is based on an observation from Fermat. Please see [5] for details and a proof.

Theorem 4.1 *If p is an odd prime then*

$$2^{p-1} \equiv 1 \ (mod\ p)$$

Now back to $p-1$. Let's assume that all its factors are less than 10,000, i.e. that it is 10000-smooth. This means that $p-1$ divides 10,000!. Using fast exponentiation (see B.2) we can calculate $m = 2^{10000!} \ (mod\ n)$ reasonably quickly. Since $p-1$ divides 10000!, we can take m modulo p to show

$$\begin{aligned} m &\equiv 2^{10000!} \ (mod\ p) \\ m &\equiv (2^{p-1})^k \ (mod\ p) \\ m &\equiv 1^k \equiv 1 \ (mod\ p) \end{aligned}$$

This means that p divides $m-1$. If n does not divide $m-1$ (and there is a good chance for this) then $\gcd(m-1,\ n)$ reveals a non-trivial factor of n. Note that any c, coprime to n, can be used in place of 2 when calculating m. Also, it is important to note that we do not know how close to 10000 we must get to factor n. We should calculate the gcd as we go along exponentiating m. As with the Pollard Rho we will perform several multiplications before calculating the gcd. Note that calculating n modulo p is equivalent to calculating it modulo n, and this is in fact what we do since the value of p is not known yet.

4.7 Quadratic Sieve

Recall Fermat's algorithm, where we searched for the difference of two squares. Maurice Kraichik generalised this idea to search for numbers whose product is a perfect square. We use this to find two numbers x and y such that $x^2 \equiv y^2 \ (mod\ n)$ but $x \not\equiv y \ (mod\ n)$, where n is a number we wish to factorise. If this is the case $\gcd(x-y, n)$ will return a non-trivial factor of n.

What we try to do is to find a large number of b-smooth numbers. These numbers can be placed in a table that contains columns for every prime in our factor base and a remainder column. Each row represents a number

$z_i = x_i^2 - n$, where n is the number to be factored and x_i is range of numbers close to $\sqrt{n}$. If this z_i is divisible by a prime in our factor base then that column contains the exponent of that prime power ($z = 96 = 2^5 \times 3^1$ would contain a 5 in the 2-column and a 1 in the 3-column; all other columns would contain 0 except the remainder column which would be 1). We remove any rows that have a remainder greater than 1 (they are not b-smooth) and then try to find dependencies between rows on the table. A dependency is several rows that when multiplied together the summed exponents for each prime will be even (this will ensure the number is a perfect square). Once we have a number of these dependencies we can get values for the x^2 and y^2 as discussed above to find a factor of n.

I omitted an initial step. The factor base should only contain prime numbers p such that n is a quadratic residue of modulo p. This means that $n \ mod \ p$ is a square number. Other numbers will not help us factorise n. When choosing the upper bound, b, to the factor base we have to bear in mind how big the table will grow. We need at least as many rows in the table as there are primes in our factor base to ensure dependencies. In fact we will usually need more because many dependencies will not lead to a factor of n.

Simply increasing the number of z-values we calculate and attempt to factorise is not good enough. Using the formula $z_i = x_i^2 - n$ will lead to bigger and bigger values as i grows. These larger numbers will be less likely to be b-smooth and their rows will be removed from the table. There are several techniques to increase the number of rows. We can accept rows that are almost b-smooth. We can use slightly different functions to derive z (this is where the Multiple Polynomial Quadratic Sieve gets its name). We can pre-multiply n by a small number to improve the factor base.

For an excellent introduction and text to this factoring method see Stephenson [26]. Also there is a good account in Bressoud [5]. I will not be implementing the MPQS - the main reason for including a description of it is to compare it with the Elliptic Curve method of factoring (at the end of this and the next chapter).

4.8 Number Field Sieve

This complex algorithm is accepted to be the best factoring algorithm. It is asymptotically faster than the MPQS. I will not go into any technical details as one can quickly get confused. I believe it was necessary to study it to complete my survey of algorithms. I will describe the five stages as set out by Huizing [10].

The initial stage is to choose two polynomials and an integer m, which is the root of both polynomials modulo n. A sieving region is also selected and a factor base bound. The second stage, the sieving is the most time consuming. Relations are pairs of integers (a, b), which factor completely over the chosen base of primes, when inputed into both polynomials. There are two forms of sieving, lattice and line, that can be applied. Although many duplicate relations are found, there is an improvement over using just one sieving method according to Lenstra et al [6].

When enough relations have been found, a filtering stage begins. Some relations are discarded and others are grouped into sets. The result is to try to reduce the amount of data. The fourth stage involves building a matrix from the remaining relations. This is likely to be very large and very sparse. An iterative approach is then taken to compress and extract the required data. The final stage uses an approximation technique to find the square root of the values from stage four. Eventually a gcd can be taken with n to reveal two non-trivial factors.

The Number Field Sieve is similar in approach to the Quadratic Sieve. There is an initial stage where things are set up, sieving and filtering stages to build a large matrix which is reduced to get the data required to factor n. A lot of storage is required during both tests. Also, it is only through experience that one knows how long to sieve for. The Swedish team that solved Simon Singh's Codebook Challenge reported [1] how they generated far too many relations using the Number Field Sieve to factor an RSA modulus. This was the final stage of the competition. The problem is, you can leave it running too long (or maybe not long enough - wasting more time). All other algorithms, including the Elliptic Curve method, return the factor as soon as they become known. The problem with these algorithms is that the answer may *never* become known. They run indefinitely. Most people agree that the sieving methods are the most realistic attacks on the RSA modulus.

Chapter 5

Elliptic Curves

Elliptic curves have become an important part of cryptography, with many cryptosystems being expressed in terms of these curves. This chapter has drawn on Bressoud [5] and Stephens [25].

5.1 What are elliptic curves?

Elliptic curves are not really elliptic, and not quite curves either! They get their name because they were used to measure ellipses and planetary orbits. They are of the form

$$y^2 = x^3 + ax + b \tag{5.1}$$

for some $a, b \in \mathbb{Z}$, where

$$4a^3 + 27b^2 \neq 0 \tag{5.2}$$

See Figure 5.1 for an example.

We define the addition of two points in the following way. Find the point where the line joining the two points intersects the curve for a third time. (Note that we use the point at infinity as a point on the curve. Therefore, any line that intersects the curve at two points, must intersect it at a third.) When we get this point its reflection in the x-axis (negate the y value) is the sum of the two points. Figure 5.2 shows the process of adding two points on an elliptic curve. Figure 5.1 describes the algorithm for doing so.

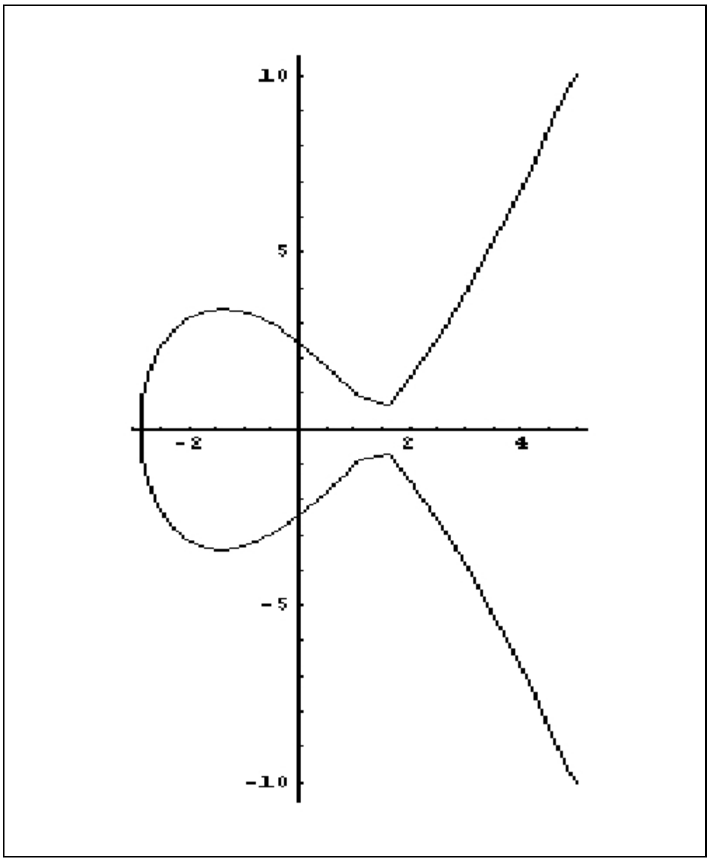

Figure 5.1: An elliptic curve

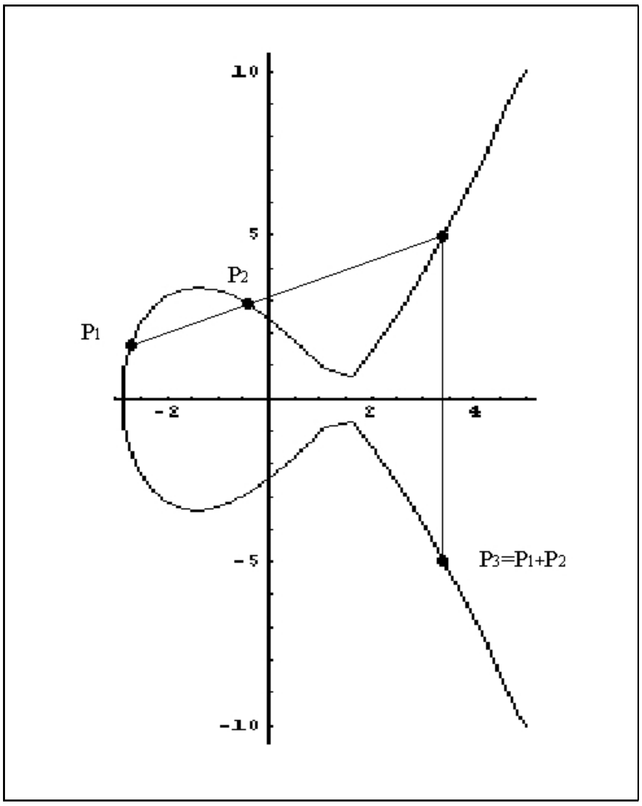

Figure 5.2: Addition of points on an elliptic curve

Let (x_1, y_1) and (x_2, y_2) be points on the elliptic curve $y^2 = x^3 + ax + b$, where $4a^3 + 27b^2 \neq 0$. We define the third point of intersection as follows. If $x_1 \neq x_2$, then

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

otherwise,

$$m = \frac{3x_1^2 + a}{2y_1}$$

Then $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ where we define x_3 and y_3 as

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m \times (x_1 - x_3) - y_1 \end{aligned}$$

Figure 5.3: Rules for addition on a curve

5.2 Groups

A group consists of a set together with a binary operation and an identity element. G = (S, ⊗, e) such that

- $\otimes$ is closed. If $x, y \in S$, then $x \otimes y \in S$.
- $\otimes$ is associative. If $x, y, z \in S$, then $(x \otimes y) \otimes z = x \otimes (y \otimes z)$.
- e is the identity. For each $x \in S$, $x \otimes e = e \otimes x = x$.
- Each element has an inverse. For each $x \in S$ there is a $y \in S$ such that $x \otimes y = y \otimes x = e$.

An example of a group is the (ℤ, +, 0) - integers over addition with zero as the identity. The inverse is the negative of the number. We can also define a group over the rationals with multiplication (ℚ, x, 1). Notice that multiplication with integers does not form a group, because it introduces the idea of rationals to form inverses. However, we can create a group using residues modulo a number, say p, to form a group. We will denote this by $(\mathbb{Z}, \times, 1)/p$.

5.3 Pollard p-1 and a General Approach to Factoring

Remember the Pollard p-1 factoring approach of section 4.6. We first formed a multiplicative group modulo p, $(\mathbb{Z}, \times, 1)/p$. The order of this group is known to be $p - 1$. The reason that we can calculate $m \ (mod\ p)$ without knowing p is because the result is correct modulo n. In reality we calculate it modulo n. When we know the order of the group we then hope that it is b-smooth (see 4.5.1) for some factor bound b. We hope that it divides $b!$ and this allows us to take the gcd with n to reveal a non-trivial factor.

The generalised technique goes as follows. We define a group modulo n. We then restrict the group modulo p, a prime factor of n. The order of this group ($|(\mathbb{Z}, \times, 1)/p|$=$p-1$ in Pollard) can then be used in the same way as in Pollard. If the order of the restricted group divides $k!$, then p divides each element of $x^{k!} - e$ (where e is the identity). We can then use the gcd algorithm with n and a member of $x^{k!} - e$ as there is a good chance that n will not divide it, thus revealing a factor.

Note that I raised x to the power of $k!$. This was only because the binary operator was multiplication in the Pollard example. When other operators are used a different notation is necessary. Bressoud [5] uses $x\#(k!)$ where the operation $\oplus$ is declared in the group. Thus $x\#3$ signifies $x \oplus x \oplus x$.

5.4 Rational Points on Elliptic curves

We can define a group based on elliptic curves. The set is the rational points on a curve including the point at infinity. The binary operation is our addition, which is closed. The point at infinity is considered a point on the curve and is also defined to be the identity for the addition operator. We denote the group of rational points on a curve, along with addition and the point at infinity as $E(a, b)$, where a and b define the elliptic curve as defined in equations 5.1 and 5.2.

Is the point at infinity the identity? Consider adding it to a given point, p, on the curve. We first draw a line between the points. This is a vertical line that passes through p. We find where this intersects the curve for a third time - and it is the reflection of p across the x-axis (the curves are symmetric across the x-axis). Finally we negate the y-value, mapping this point back onto p. Therefore the point at infinity added to any other point on the curve gives that point under our definition.

Restricting the group modulo an integer is the next step. We denote the restricted group modulo p as $\mathrm{E}(a, b)/p$. Finding the order of this group is difficult for large primes. However we do know that the order lies in a certain interval. The following theorem is due to Helmut Hasse, published in 1934.

Theorem 5.1 *The order of E(a,b)/p lies in the interval*

$$I(p) = (p + 1 - 2\sqrt{p},\ p + 1 + 2\sqrt{p}$$

The interesting thing about this type of group is that by changing the values of a and b, that is choosing new elliptic curves, we define different groups modulo p. Each of these groups will have a different order, and hopefully we can find an order that is b-smooth for a practical bound b. In the Pollard $p-1$ test we only had one group and one order. The test was only successful when $p-1$ was b-smooth. This gives the elliptic curve method a clear advantage over Pollard. If things aren't working out with the curve we are using, then simply choose another.

To find a prime divisor q of n we need to have some integer in the interval $I(q)$ divide $K!$, where K is a bound on the factors of the order of the group. It is conceivable that a curve could be found for such a number. But it is still quite difficult to find a curve based on a and b for a given order chosen because it is b-smooth. The technique we shall use is to randomly generate elliptic curves until we find one that fits.

5.5 Elliptic Curve Factorization

We need to be able to calculate points on elliptic curves quickly. Given a starting point r on a curve $\mathrm{E}(a, b)$ we must calculate $r\#k!$. When we reach the order of $\mathrm{E}(a, b)/p$ we expect that $\gcd(x, n)$ will return a non-trivial factor of n. In practice, we only need to worry about the first coordinate of r. The following formula allows the next x-coordinate to be calculated without the need for the y-value. Let (l, m) be the result of $(x, y)\#2$, if $y \neq 0$ then,

$$l = \frac{(x^2 - a)^2 - 8bx}{4(x^3 + ax + b)}$$

All x-coordinates $(x, y)\#2i$ can be found like this. To find the x-coordinate of $(x, y)\#(2i+1)$ we base it on knowing $(l, m) = (x, y)\#i$, $(r, s) = (x, y)\#(i+1)$

and $(u, v) = (x, y)\#(2i + 1)$ as long as $l \neq r$ and $x \neq 0$ then

$$u = \frac{(a - lr)^2 - 4b \times (p + r)}{x \times (p - r)^2}$$

Using this trick, we decrease the time taken to generate points.

5.6 Advantages

The Elliptic Curve method is a generalisation on the Pollard $p - 1$ method. No longer are we constrained to situations where $p - 1$ is a smooth number. But the algorithm generally only works well with smaller values of p. This is because as p gets smaller, the range $I(p)$ covers smaller numbers, and it is easier to find a smooth number with a curve in this interval. However the EC method has other advantages over sieving techniques.

Programming an implementation of the EC algorithm is comparitively easy. Care needs to be taken that the language deals with large numbers correctly (Mathematica is good at this). Both the MPQS an GNFS have many complicated stages. GNFS is quite recent and much of the necessary information needed to build an implementation is divided amongst several papers. The polynomial selection stage of the GNFS is quite complicated and care has to be taken otherwise the other stages may take longer than necessary. The final stages of the both MPQS and GNFS benefit greatly from a supercomputer that can crunch the numbers searching for dependencies (the sieving stages having been done on a distributed network for example). A Cray C916 supercomputer was used during the final stages of the RSA-155 attempt. These sort of resources are not available to many people.

The storage requirements of this algorithm are very lightweight. This is in contrast to both the quadratic and number field sieves. The advantage of this is that we can compute on many different curves in parallel. Using a distributed system the sieving stages can be performed on a large number of ordinary computers, but the tables generated have to be collected once complete. In the RSA-155 attack, 124.7 million relations were collected from twelve different sites. With an elliptic curve attack however, each computer would be given a selection of curves to check. When they had been covered, they could request more curves from a centralised database. Alternatively, the programs could generate their own random curves.

However it is unlikely that the EC method will ever out-perform the number field sieve.

Chapter 6

Attacks on RSA

Besides attacking the key by trying to factorise the public modulus, there do exist several attacks which are as intractable as factorization, or depend on the users not using the system correctly. Some others depend on the attacker being able to trick the victim through some form of social engineering. All these are valid ways of attacking the RSA, and have to be included in any security plan for an organisation.

6.1 Exhaustive Search

Exhaustive searches mean testing every possible key in the key-space. One way to do this would be to see what exponent would return a ciphertext to the original message. Once the private exponent has become known to the attacker, then they could factor the modulus easily. Rivest shows in the original paper [19] that this is possible. I used an algorithm from the Handbook of Applied Cryptography [15] that quickly retrieves the factors even for large key lengths. Rivest suggested that this method of attack is equivalent to factoring and that it is at least as difficult as the factorization problem. Therefore it should be compared to other factoring algorithms, and it works much slower than other algorithms.

RSA can be used on smart cards. The idea being that the private key is stored on the card. Whenever a signature is required the card is inserted into a reader that powers the card. The card itself does the signing or encrypting work. This has lead to the suggestion that a small sized private key should be used. This will speed up the transaction (smart cards work much slower than normal computers in general). However by doing so one reduces the search space considerably. There is an increased risk that an attacker could find the private exponent, knowing that it is of the smaller smart card kind. However, even in this circumstance, an exhaustive search is not the best attack. The following attack retrieves the secret faster.

6.2 Low Decrypting Exponent

Wiener [29] describes an attack on RSA which works when the decrypting exponent has less than a quarter the number of bits as the modulus. Later Boneh and Durfee [4] extended the range of the attack to bound d to less than $n^{0.292}$. At ASIACRYPT '99 Sun, Yang and Laih [28] proposed three RSA variants which defeated the previous attacks. However, Durfee and Nguyen [9] extended the Boneh-Durfee attack to defeat two of these suggested systems. The work is continuing, the aim to extend the range of the attack to $n^{0.5}$.

The reason I have cited this number of papers is to show how important research into short private keys is. I will restrict my description to Wiener's original attack. There is an algorithm that uses continued fractions to find a numerator and denominator of a fraction given a close enough estimate of the fraction. We will call this the continued fraction algorithm. Continued fractions are expressions in the following form.

$$\cfrac{a_1}{q1+\cfrac{a_2}{q2+\cfrac{a_3}{\ddots\cfrac{}{q_{m-1}+\frac{a_m}{q_m}}}}}$$

We call the q_i parts the quotients. We normally convert a continued fraction to a fully expanded form, where each of the a_i is equal to 1. This means the structure can be represented by a list of the quotients $< q_0, q_1, ..., q_m >$. A positive rational, f, can be converted to a continued fraction easily by subtracting away the integer part of f and inverting the remainder and then subtracting the integer part until the remainder is zero. It is easy to reconstruct a rational from an expanded form. By using only part of the expanded list we get a slightly different rational. We will use this to guess/estimate the rational we are searching for.

Using the information that we know from the public key (e and $n = pq$), it is possible to construct the following rational

$$\frac{e}{pq} = \frac{k}{dg}(1 - \delta), \qquad \text{where } \delta = \frac{p + q - 1 - \frac{g}{k}}{pq}$$

This is a close underestimate of $\frac{k}{dg}$. We will be able to calculate this fraction using the continued fraction algorithm if δ is small enough. The paper suggests that this will be possible whenever

$$kdg < \frac{pq}{3/2(p + q)}$$

The continued fraction algorithm goes as follows. We begin with a rational, f' which is an underestimate of f. Repeat the following steps until f is found.

1. Generate the next quotient, q_i of the expansion of f'.

2. Reconstruct the fraction to $< q_0, q_1, ..., q_i + 1 >$, if i is even, or $< q_0, q_1, ..., q_i >$, if i is odd.

3. Perform a test to see if this fraction is equal to f

Step 2 ensures that the guess is larger than f'. The test we use in the RSA version of the algorithm takes the guess of $\frac{k}{dg}$ calculated in step 2 and attempts to fill in the rest of the variables for the equation, eventually revealing d.

6.3 Common Modulus

One of the early weaknesses found was in a system of RSA where the users within an organisation would share the public modulus. That is to say, the administration would choose the public modulus securely and generate pairs of encryption and decryption exponents (public and private keys) and distribute them to all the employees/users. The reason for doing this is to make it more convenient to design, implement and manage software for such a system.

However, Simmons [22] shows how this would allow any eavesdropper to view any messages encrypted with two keys; for example when a memo is sent to several employees. DeLaurentis [7] went further to demonstrate how the system was at even more risk from insiders, who could break the system completely, allowing them to view all messages and sign with anybody's key. Of course Rivest had already shown this sort of a system was insecure by describing how anyone could factor the modulus if they had a pair of encrypting and decrypting exponents for that modulus. However despite the stupidity of using such a system, the Simmons attack is necessary to perform the Faulty Encryption attack of section 6.4.

The Simmons attack works when the cryptanalyst receives two ciphertexts, c and c', created from one unknown message, m, encrypted with two exponents, e and e', and one common modulus n. Their relation is $c \equiv m^e (mod\ n)$ and $c' \equiv m^{e'} (mod\ n)$. If $\gcd(e, e') = 1$ then we can use the Extended Euclidean Algorithm (see Appendix B.1.2) to get the equation

$$ae + be' = 1$$

We can then retrieve m from

$$c^a c'^b = (m^e)^a (m^{e'})^b = m^{ae+be'} (mod\ n)$$

But since $ae + be' = 1$, we get $m^{ae+be'} = m^1 = m$. The actual implementation has a few extra details; please see the code for details.

6.4 Faulty Encryption

Joye and Quisquater showed [14] how to capitalise on the common modulus weakness due to a transient error when transmitting the public key. Consider the situation where an attacker, Malory, has access to the communication channel used by Alice and Bob. In other words, Malory can listen to anything that is transmitted, and can also change what is transmitted. Alice wishes to talk privately to Bob, but does not know his public key. She requests, by sending an email, to which Bob replies. But during transmission, Malory is able to see the public key and decides to flip a single bit in the public exponent of Bob, changing (e, n) to (e', n).

When Alice receives the faulty key, she encrypts the prepared message and sends it to Bob (Malory also gets it). But of course, Bob cannot decrypt it because the wrong key was used. So he lets Alice know and they agree to try again, starting with Bob re-sending his public key. This time Malory does not interfere. Alice sends the message again, this time encrypted with the correct public key.

Malory now has two ciphertexts, one encrypted with the faulty exponent and one with the correct one. She also knows both these exponents and the public modulus. Therefore she can now apply the common modulus attack to retrieve Alice's message, assuming that Alice was foolish enough to encrypt exactly the same message the second time. The hope is that Alice will not suspect any foul play because the mistake looked like a transmission error.

6.5 Cycle Attack

This attack is due to Simmons and Norris [23]. The idea is that we encrypt the ciphertext repeatedly, counting the iterations, until the original text reappears. The reason for this is that the encryption operation creates a permutation of $\mathbb{Z}_n$. Thus each number, repeatedly encrypted will cycle back to itself, eventually. The number of re-cycles will decrypt any ciphertext block. This method is very slow and for a large key it is not a practical attack. The reason is the size of the permutation cycle, i.e. the search space, is too big.

A generalisation of the attack due to Williams and Schmid [31] allows the modulus to be factored and it works faster the majority of the time. But even this will still have difficulty when a large key is used. The generalised form does tries to find the least value of k for which

$$\gcd(n,\ c^{e^k} - c)\ >\ 1.$$

If this occurs then we have most likely factored n. If not, then we can use the number of re-cycles to decrypt any message anyway.

The bottom line is that the generalised form of the cipher attack is another factoring algorithm. It is not efficient, and therefore the attack is not good enough compared with modern factoring algorithms (e.g. Number Field Sieve). Also the use of p^{--} strong primes (see Appendix A.2) aids the security.

I noticed an improvement on this algorithm. The suggested way is to use the public exponent of the public key to re-encrypt the text. However any exponent should work so long as it is coprime to $(p-1)(q-1)$ (where p, q are factors of the modulus). So I suggest using an exponent such as $2^{16}+1$. This number has only two '1's in its binary representation. Using binary fast exponentiation (see Appendix B.2), we use only 16 modular squarings and 1 modular multiplication. This is likely to be faster than the actual public exponent. The trouble is that we cannot be sure that it is coprime to $(p-1)(q-1)$. In practice, many RSA systems use $2^{16}+1$ as the encrypting exponent for its speed. Even with this improvement the system is too slow to be a worthwhile attack.

6.6 Low Encrypting Exponent

In the cycle attack section above, I suggested that the encrypting exponent could be chosen to make the system more efficient. Many RSA systems use

$e = 3$ to make encrypting faster. However, there is a vulnerability with this attack. If the same message is encrypted 3 times with different keys (that is same exponent, different moduli) then we can retrieve the message.

For example, supposing we had 3 public keys which each use 3 as the exponent: $(3, n_1)$, $(3, n_2)$ and $(3, n_3)$. A message, m, is now encrypted into three different ciphertexts $c_i = m^3 \ (mod\ n_i)$. The Chinese Remainder Theorem (see Appendix B.4) allows us to find a positive solution $x < n_1 n_2 n_3$ to the congruences

$$\begin{aligned} x &\equiv c_1 \ (mod\ n_1) \\ x &\equiv c_2 \ (mod\ n_2) \\ x &\equiv c_3 \ (mod\ n_3) \end{aligned}$$

Since we know that m^3 is also less than the product of the moduli, we know that $x = m^3$ and therefore retrieve the message by calculating the cubed root of x.

To protect against this attack one can "salt" the message by adding random bits to each block of the message. A larger encrypting exponent can also be used such as $2^{16} - 1$. This is often used because it is faster than most exponents (see Section 6.5).

6.7 Adaptive Ciphertext

What happens when you decrypt a message but all that appears is garbage, seemingly random characters? Should you forward the message down to the technical support people or ask for a retransmission of the message? Was this a result of noise on the communication channel?

In Section 6.4 we showed that Malory could tamper with the transmission of the public key and eventually recover a message by tricking Alice and Bob. Malory could also however alter the ciphertext itself by multiplying it by a number. She could generate a random number x and get a corrupted ciphertext $c' \equiv c \times x^e \ (mod\ n)$, where c is the correct ciphertext for the message m encrypted with (e, n), the public key of the receiver, Alice. When Alice gets the message she will attempt to decrypt it, producing

$$y \equiv c'^d \equiv (cx^e)^d \equiv c^d x^{ed} \equiv mx \ (mod\ n)$$

The message, m is retrieved by multiplying by the multiplicative inverse of x modulo n (see Appendix B.1.2).

This attack depends on being able to get the decrypted message. If Alice does not realise she is under attack, she may send the message to technical support to complain. But Malory might be monitoring the channel. Alice may not realise the importance of the message and fail to properly delete the output (this could also be a technical problem due to a poor implementation). Joye and Quisquater [13] give a discussion on this attack and on how the same attack works on other cryptosystems.

A modified version of this attack is where Malory persuades Alice to sign a message of hers. However this is a message that was originally addressed to Alice but has been intercepted and altered in the same way as above. If Alice signs the message then Malory can recover the original. There are many similar attacks that require that the attacker can access the decrypting process of the victim once.

Chapter 7

Implementation and Results

My first step in this project was to attack the student codes problem as I have described in Chapter 3. Then I started to work on implementing RSA in Mathematica. I needed to do this to create a system of functions that would enable me to generate various test cases. My intention was to develop a set of notebooks that could be read through, giving an explanation along the way.

7.1 Prime Number and Key Generation

I allowed the user to specify the number of decimal digits required in the primes to be generated. I also provided the option for strong primes (see A.2). The default is to use p^{--}-strong primes. I had to create a simple key management system to keep the keys generated in good order. I made use of Mathematica's pattern capabilities to do so.

7.2 Encoding and Encrypting

Encoding involves changing a string of characters into blocks of numbers. The program uses the standard ASCII numbers for the character set. It first calculates the number of blocks required (based on the length of the public modulus), and then shifts the first character to the left eight binary digits and then appends the next, until a block is completely filled. This may become more obvious from studying the code. Decoding simply reverses this.

The encryption function takes a string, converts to a list of blocks, and then maps Mathematica's fast exponentiation function to each block using the key specified. The resulting ciphertext is the input for the decryption algorithm which calls on the decoding algorithm. I made sure to use abstraction to make it clear what function does. Abstraction is the technique to remove complicated parts of a function to a separate function that can be called by the first.

7.3 Knowing a Decrypting Exponent

This attack works quickly and easily. Given both the public and private key one can quickly derive the factorisation of p and q. However, searching for the decrypting exponent, given only the public key, is extremely slow. Using a tiny 25-bit key the search took 12 minutes. Compare this to a factoring algorithm that factored the modulus in less the 5 hundredths of a second. The Cycle Attack is a better way to search for the decrypting exponent (but not much better). In my notebooks the algorithm took less than 2 seconds to retrieve one of the factors of a 847-bit modulus. Incidentally this is faster than the time it took to generate the key in the first place.

7.4 The Cycle Attack

The Cycle Attack was described in Chapter 6.5. Running time for this algorithm is quite poor. The following table show typical results. Note that I never got anywhere with keys over 75 bits. The results in Table 7.1 show that p^{--} are resistant to the attack more often than normal random primes at this level.

Timing results are difficult to get accurate results on. The number of re-

No. of Bits	Normal Primes	String Primes
<25	Always Works	Always Works
<50	Usually Works	Rarely Works
<75	Rarely Works	Never Works
>75	Never works	Never Works

Table 7.1: Chances for Cycle Attack to Work

cycles required grows exponentially with the number of bits in the modulus. Add this to the fact that a recycle with a larger modulus takes longer. I tried one million recycles on a 60-bit modulus to get an idea. It took more than fifteen minutes but found no factors.

7.5 Low Encrypting Exponent

This attack was very easy to implement. I simply used the algorithm mentioned in Appendix B.4. Once complete it was easy to run and test. I simply generated three keys using the modified RSA key generator, that allows you to specify the exponent - in this case 3. I then encrypted a message three times and ran Guass's algorithm. The resulting number is the message block raised to the power of three. I tested with different exponent, the key idea being that the number of messages needed is equal to the low exponent that is being attacked.

7.6 Low Decrypting Exponent

The algorithm for this attack is quite complicated and tricky. But I managed to get a version working. To do so I needed to become versed in continued fractions. I remember Micheál teaching us about them in first year maths, but little more than that. My problems were compounded when I introduced a bug into my RSA key generation code. It was a while before I noticed the bug, and I wasted time trying to debug the wrong piece of code. The algorithm usually works better when the decrypting exponent has a much lower number of bits than the theoretical limit, a quarter the number of bits as the modulus.

7.7 Other Attacks

The Common Modulus and Faulty Encryption (Chapter 6) attack don't really need results. The results are pretty instantaneous. The trouble is in how to gain all the information needed from the victims. Some ability to manipulate and fool people (social engineering) is necessary. The Adaptive Ciphertext attack falls into this category as well.

7.8 Factoring Algorithms

I implemented Trial Division, the Pollard Rho and $p-1$ algorithms, Fermat's algorithm and also the Elliptic curve method of factoring. Before I describe the work, perhaps it would be a good idea to examine the cases where these and other algorithms work well. We will create four classes for examination.

Size of p and q

Algorithms such as Trial Division and the Pollard Rho test fall into this class. They will generally only work with low values of p or q. The Elliptic curve method also falls into this class.

Smoothness of $p-1$ or $p+1$

The Pollard $p-1$ attack will work when $p-1$ is b-smooth. There also exists another test due to Williams that allows n to be factored if $p+1$ is smooth. I have not discussed this attack. It is based on using Lucas sequences instead of raising to powers. For more details see Bressoud [5].

Closeness of p and q

Fermat's algorithm works well when the factors are close to each other and to the square root of n.

Size of n

Algorithms such as MPQS and the General Number Field Sieve fall into this category. They require a large amount of memory and powerful number crunchers.

7.9 How my implementations fared

None of the factoring techniques I used would stand up to a really large modulus. Some reasonable test was needed to provide a decent challenge. The RSA Partition Challenge (mentioned in Chapter 2.1) provided such a test. These numbers are a mix between trivial, medium and difficult factoring tasks. By trivial I mean that trial division will succeed faster then any other technique. By medium, I mean that there are some large factors, but one of my algorithms (Pollard rho, $p-1$ or elliptic curves) would find the factor. Difficult numbers were one with two very large primes that are very close to RSA type moduli. All of the challenge numbers have been completely factored, making it easier to proceed. I did not waste time on numbers that I clearly could not hope to factor.

7.10 A Novel Approach to Factoring

During the year I wrote a report entitled *Programming Paradigms in Amorphous Computing*. An amorphous computer is made up of an enormous number of processors, that can communicate only to their close neighbours. The size of each processor should be close to that of a grain of sand. These tiny computers will be mass produced so as to reduce expense. The idea is that fault tolerance will be achieved through redundancy and not by quality control. This will reduce costs also. Amorphous computers can be mixed with bulk materials, such as paint, concrete and gels, to become part of a structure. There are simple paradigms which allow the processors to locate themselves within the structure, to organise themselves into groups and to divide workloads. One possible use is a bridge that can report on stress or traffic load. But they could easily play a part in cryptanalysis.

The fact is that we are on the verge of this new technology. We can build such tiny circuits and the programming techniques are already being developed to run them. There are some issues outstanding, such as how to provide power throughout the structure, which must be resolved. But imagine if a large organisation was to build their building with concrete that contained the amorphous computer. Imagine that it then programmed the computer with a factoring algorithm and daily passed it RSA keys that it wanted to break

I described in Chapter 2.3 the means by which the DES-III challenge was broken in 22 hours. The combined computing power of Distributed.net did an exhaustive search on the 2^{56} possible keys. Over 100,000 workstations were involved. Consider a sky-scraper that also functions as an amorphous computer. There is potential for millions of processors to be used. A brute force attack may not in fact be out of the question! I have included a copy of my report on the CD-ROM. It contains many references to current work on this topic.

Chapter 8

Conclusions

8.1 Lessons to be Learned

I would like to sum up some of the observations that I have made. Firstly, is that RSA is as difficult to break as factoring. This has not been formerly proven, but for the time being, it is a statement of truth. Given the best factoring methods, no one should use a 512-bit modulus for security. An organisation that is willing to invest several million in hardware and give a few months to crack a key could do so. 1024-bit keys are safe at the moment, however over the next 20 years, we must move towards 2048-bit keys. The rate of progress in number theory and factoring in particular has been faster than expected. Rivest did not believe in 1977 that a 512-bit number would ever be factored.

Small encrypting or decrypting exponents should not be allowed. This is bad news for smart cards. The theorectical limit is that d must be greater than $N^{0.297}$ but it would be dangerous to believe that this limit is fixed. A modification of the technique discovered over the next few years could raise this to as high as $N^{0.5}$. It has already been increased from $N^{0.25}$.

Always destroy decrypted messages, especially ones that look like garbage. You never know when you are being attacked. Software implementations must ensure that they do not "leak" information that appears to be useless.

One man's garbage is another man's treasure.

Never encrypt the same message twice. Use random padding of blocks. This is an excellent tactic as it will really hamper cryptanalysis. However, the pseudo-random number generator should not allow the cryptanalyst to guess or predict the numbers that will be generated.

The most important thing of all is to be aware that there are new developments being made all the time. The cryptographic industry does not stand still. There are three Advances in Cryptography proceedings per year in America, Europe and Asia. People who use cryptographic products need to be aware of what is going on in these conferences, and not just the implementors of the software. I was surprised to find that some financial institutions probably still use 512-bit keys. Some even use a low encrypting exponent. Not to mention that most credit card transactions over the internet use the minimum key size. One way to keep in touch is to subscribe to Bruce Schneier's monthly Cryptogram newsletter. This can be done from the counterpane.com web-site.

8.2 If I had more time

If I was only starting the project now, with all the knowledge I have gained, I would attempt to implement one of the sieving algorithms. My intention would be to factor an RSA challenge number. RSA-150 for example has not been solved. This would be a very ambitious task, and would rely on the Computer Science Deparment allowing me to use their resources.

In my course Computer and Internet Security, I implemented in Java, the Advanced Encryption Standard (AES). I would have liked to include some of what I learned about symmetric algorithms from this. Unfortunately I did not integrate it into project. However, I plan to add the Java applet to the CD-ROM.

The rest of the chapter discusses the skills used and learned during the course of the project. The Personal and Career Development Record helped me to express these skills.

8.3 Programming with Mathematica

I have used *Mathematica* since my first year as an undergraduate, but it was this year that I began to finally use it for more concrete work. The course has focused on using Java to write programs, but we have also been shown functional programming languages such as Lisp. I learnt how to use patterns to create structers within the language. I made use of as much in-built code as possible.

I intended my notebooks to be easy to read through, to make sense from a scientific paper. I found it difficult sometimes while reading papers to visualise the algorithms and systems. I hope that by reading the notebooks and trying out some of the calculations, understanding will be easier.

One problem I came across was the lack of memory. Within the interactive Mathematica environment, it is very easy, for example, to generate a list of primes. So easy that on occasion, I didn't stop to consider the size of the list. I lost quite a bit of work through system crashes because of such careless actions. Actually, Mathematica is quite bad with regard to loss of work. It does not have an automatic backup system. The undo function is very inconsistant. If a paragraph is deleted accidentally, the undo does not get it back. But if something is typed and undo is called, it undoes more text than might be expected, and *that* piece of text is gone for good. It can be quite frustrating to use the program for a long period of time. It does teach one to save regularly though.

8.4 LaTeX

This report has been written up using the LaTeX typesetting program. Many of my lecturers and also my supervisor suggested using it, particularly for the ease at which mathematical functions can be written. I downloaded and burnt the TeX-Live CD from the Internet and ran it on my Windows PC.

From the beginning I was impressed with the focus that is put on the logical planning of the document. So often using a WYSIWYG interface, one has to be more concerned with appearance, and overlooks the logical layout. Being more or less freed from worrying about how the end product will look allowed me to concentrate on building a complete, planned report.

Building mathematical statements is intuitive in LaTeX. Again structure is the all important factor - the program will sort the appearance out itself. The report contains many mentions of variables such as p and n. Being able to quickly emphasise these characters spared a lot of work (it only takes two extra characters).

Building a correct bibliography is quite time consuming. The BibTeX package allows the construction of a library type file containing all references used. When citations are made within the document, BibTeX extracts the data and formats it correctly. It also helps by warning that there are missing fields. I found another program on the Internet that converts BibTeX files into a range of other files, including a html format. I used this output to create the index to papers that is on the CD-ROM.

One of my favourite tricks in LaTeX is the referencing commands. A chapter, section, table, figure or even a paragraph can be labeled with a name. Then it is easy to refer back to it. As chapters and sections change order, LaTeX takes care of re-numbering all the references. It also allows the building of a table of contents without any effort. Programs such as MS-Word contain similar functionality, but I find that it is much harder to know what is going on when all the details are hidden by a fancy user interface.

A problem many people face when first using LaTeX is that it can be difficult to debug. The cryptic messages from the compiler are sometimes misleading. Fortunately I have a lot of experience dealing with such compiler-like output (from programming in Java and C) and had no touble here. I was able to adapt easily to the new system and "decrypt" the error messages - I am after all a hacker.

8.5 Groups and Elliptic Curves

In first year we were told about groups, rings and fields. Little did I know that I would be going back to study these topics ever again. The second time around, however I was able to appreciate how much I already knew. My background knowledge allowed me to get to the bottom of the Elliptic Curve method.

8.6 Researching Skills

Perhaps my most improved skill is my ability to research, to track down papers and articles. Another skill is to be able to discard the articles that are not of interest. Before fourth year I would rely only on altavista or other search engines. However I located more sophisticated tools that allow a great number of papers to be retrieved, or at least located. I spent the first few months of the project in the library searching through the periodicals. The Springer web-site allows its publications to be searched - the best way to get the contents of their publications. Also, many of the documents are available to download from the site. Citeseer.com allows researchers to locate documents that have cited an original document. This means that a trail can be followed from one paper to many related papers.

8.7 Complexity Analysis

When deciding what algorithms we want to use to solve a problem, it is important to take the complexity analysis into account. We say that one algorithm is asymptocically faster than another if its speed improves with an increase in the input size. We would like to find polynomial time solutions to decision problems. This type of algorithm are faster than exponential algorithms such as the factoring algorithms. The interesting thing about it is that should a polynomial time algorithm for factoring be discovered, then it may lead to solutions for other difficult problems. I have not included any such analysis in the report - the references cited provide better descriptions - but I did have to be able to understand these articles.

8.8 Writing

Although many of the projects over the four years in Trinity have required written reports, most of the work goes into coding and implementation. It's very easy to forget how to write well-structured sentences. I studied Latin for the Junior Certificate and I found that this helped my English grammar, especially in structuring sentences. But I have found in writing this report that I no longer have the same ability. I find myself in doubt about things that I was once confident of. However, while writing such a large amount of English, I have noticed that things were improving. Thanks to my spell checker I have standardised to UK spelling. This may not seem very important, but during an early proof-read, it was disconcerting to see the American 'z' words intermixed with the correct 's' version (factorizing instead of factorinsing). It's hardly surprising, as the majority of papers that I read originate in the USA.

Glossary

Advanced Encryption Standard (AES) is the newly standardised symmetric key algorithm. It allows key-sizes and block lengths of 128, 196 or 256 bits. It is a symmetric cipher.

Ciphertext is the encrypted form of a message. It can be decrypted into a plaintext.

Common Modulus is a flawed implementation of RSA in which members within an organisation share a modulus but use different exponents in their public and private keys.

Cycle Attack By repeatedly encrypting the encrypted message, the original message will eventually return.

Data Encryption Standard (DES) was developed by IBM in the 1970s. Most believe its 56-bit key too short. It is a symmetric cipher.

Key-space is the number of possible keys for an encryption system. For example DES has 2^{56} keys.

Known Plaintext Attack: When the cryptanalyst has access to a plaintext-ciphertext pair.

MPQS: The Multiple Polynomial Quadratic Sieve factoring method.

NFS: The Number Field Sieve factoring method.

Non-Trivial Factors of n are divisors of n that are not equal to n or to 1.

Plaintext is a message before it has been encrypted, or the message recovered from a ciphertext.

Private Keys are used to decrypt messages in a public key cryptosystem. They can also be used to create signatures.

Public Key Cryptosystems or asymmetric cryptosystems use different keys to encrypt and decrypt. The public key should be published as widely as possible. The private key is kept secret.

Public Keys are used to encrypt messages in a public key cryptosystem. They can also be used to verify signatures.

Smooth - We say that a number, n, is B-smooth if all of its prime-power factors are less than the factor bound B.

Symmetric Ciphers (or secret key ciphers) were the predecessors to Public Key Cryptosystems. Both sender and receiver must know the secret key.

Bibliography

[1] Fredrik Almgren, Gunnar Andersson, Torbjorn Granlund, Lars Ivansson, and Staffan Ulfberg. How we cracked the code book ciphers. http://answers.codebook.org, Oct 2000.

[2] George E. Andrews. *The Theory of Partitions*, volume 2 of *The Encyclopedia of Mathematics and its Applications*. Addison-Wesley, 1976.

[3] Derek Atkins, Michael Graff, Arjen K. Lenstra, and Paul C. Leyland. The magic words are squeamish ossifrage (extended abstract). In *Advances in Cryptology – ASIACRYPT '94*, volume 917 of *Lecture Notes in Computer Science*, pages 263–277, Wollongong, Australia, Dec 1994. Springer-Verlag.

[4] Boneh and Durfee. Cryptanalysis of RSA with private key d less than $n^{0.292}$. In Preneel [18].

[5] D. M. Bressoud. *Factorization and Primality Testing*. Springer-Verlag, New York, 1989.

[6] M. Cavallar, B. Dodson, A. Lenstra, W. Lioen, P. Montgomery, B. Murphy, H. Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, and C. Putnam. Factorization of a 512-bit rsa modulus. In Stern [27], pages 1–18.

[7] John M. DeLaurentis. A further weakness in the common modulus protocol for the RSA cryptoalgorithm. *Cryptologia*, 8(3), Jul 1984.

[8] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov 1976.

[9] Glenn Durfee and Phong Nguyen. Cryptanalysis of the RSA schemes with short secret exponent. In Okamoto [16], pages 14–29.

[10] M Elkenbracht-Huizing. An implementation of the number field sieve. *Experimental Mathematics*, 5(3):382–394, Jul 1996.

[11] Dan Farmer and Wietse Venema. Improving the security of your site by breaking into it. ftp://ftp.porcupine.org/pub/security/admin-guide-to-cracking.101.Z, Dec 1993.

[12] M. Gardner. Mathematical games. *Scientific American*, 237(2):120–124, Aug 1977.

[13] M. Joye and J. Quisquater. On the importance of securing your bins: The garbage-man-in-the-middle attack. In *4th ACM Conference on Computer and Communications Security*, pages 135–141. ACM Press, 1997.

[14] M. Joye and J.-J. Quisquater. Faulty RSA encryption. *UCL Crypto Group Technical Report CG-1997/8*, 1987.

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

[16] Tatsuaki Okamoto, editor. *Advances in Cryptology - ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, Kyoto, Japan, Dec 2000. Springer.

[17] C. Pomerance, J. Selfridge, and S. Wagstaff Jr. The pseudoprimes to 25×10^9. *Math. Comp.*, 35:1003–1026, 1980.

[18] B. Preneel, editor. *Advances in Cryptology - Eurocrypt 2000*, volume 1807 of *Lecture Notes in Computer Science*, Bruges, Belgium, May 2000. Springer.

[19] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[20] R. Rivest and R. Silverman. Are strong primes needed for rsa, Dec 1998.

[21] R. L. Rivest. Finding four million random primes. In *Advances in Cryptology, Crypto '90*, Lecture Notes in Computer Science, pages 625–626, New York, 1990. Springer-Verlag.

[22] Gustavus J. Simmons. A 'weak' privacy protocol using the RSA crypto algorithm. *Cryptologia*, 7(2), Apr 1983.

[23] Gustavus J. Simmons and Michael J. Norris. Preliminary comments on the M.I.T. public-key cryptosystem. *Cryptologia*, 1(4):406–414, Oct 1977.

[24] Simon Singh. *Fermat's Last Theorem.* Fourth Estate, 6 Salem Road, London W2 4BU, 1997.

[25] N.M. Stephens. Lenstra's factorization method based on elliptic curves. In Williams [30], pages 516–522.

[26] David Stephenson. Factoring very large numbers. Available to download at citeseer.nj.nec.com.

[27] Jacques Stern, editor. *Advances in Cryptology - EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, Prague, Czech Republic, May 1999. Springer.

[28] Hung-Min Sun, Wu-Chuan Yang, and Chi-Sung Laih. On the design of rsa with short secret exponent. In *Advances in Cryptology - ASIACRYPT'99*, Lecture Notes in Computer Science, pages 150–164, Singapore, Nov 1999. Springer.

[29] Michael J. Wiener. Cryptanalysis of short rsa secret exponents. *IEEE Tran. Inform. Theory*, 36(3):553–558, May 1990.

[30] H. C. Williams, editor. *Lecture Notes in Computer Science*, volume 218. Springer, 1986.

[31] H. C. Williams and B. Schmid. Some remarks concerning the M.I.T. public-key cryptosystem. *BIT*, 19(4):525–538, 1979.

Further Reading

This is a list of further reading that was not directly referenced from the text.

- Simon Singh. *The Code Book* Fourth Estate, 6 Salem Road, London W2 4BU. 1999. ISBN: 1 85702 889 9

- Don Coppersmith, Matthew Franklin, Jacques Patarin, and Michael Reiter. Low-exponent RSA with related messages. In *Advances in Cryptology - CRYPTO '96*. Springer-Verlag, 1996.

- G. Davida. Chosen signature cryptanalysis of the RSA (MIT) public key cryptosystem. Technical Report TC-CS-82-s, Dept. of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, USA, Oct 1982.

- Dorothy Elizabeth Denning. *Cryptography and Data Security.* Addison-Wesley,1983.

- Y. Desmedt and A. M. Odlyzko. A chosen text attack on the RSA cryptosystem and some discrete logarithm schemes. In H. C. Williams, editor, *Lecture Notes in Computer Science*, pages 516-522, volume 218. Springer, 1986.

- Dorothy E. Denning. Digital Signatures with RSA and Other Public-Key Cryptosystems. *Commun. of the ACM*, 27(4):382-392, Apr 1984.

- A. Lenstra, H. Lenstra, J. Manasse, and J. Pollard. The number field sieve. In *Proc. 22nd Annual ACM Conference on Theory of Computing*, pages 564-572, Baltimore, Maryland, May 1990.

- Ronald L. Rivest. Remarks on a proposed attack on the M.I.T. public-key cryptosystem. *Cryptologia*, 2(1):62-65, Jan 1978.

- Ronald L. Rivest and Adi Shamir. How to expose an eavesdropper. *Commun. of the ACM*, 27(4):383-395, Apr 1984.

- Adi Shamir. How to Share a Secret. *Commun. of the ACM*, 22(11):612-613, Nov 1984.

Appendix A

RSA - How it works

This appendix provides a brief description of RSA key generation.

A.1 Basic Generation

To generate a key, choose two very large prime numbers. The primes should be about half the size of the desired modulus. It is a good idea for them to differ in length by several digits. There is a simple algorithm to generate such primes. Use a pseudo random number generator to get a random number of the required size. Test it for primality (using a strong pseudoprime test as described in Appendix B.3.2). If it is not prime then test the following number, and continue until a prime is found. Primes are well distributed, so this test should not take long to complete.

Once the primes p and q have been chosen, we have the public modulus, $n = p \times q$. The encrypting exponent, e, can be chosen randomly or arbitrarily. The one condition is that the $\gcd(e, (p-1) \times (q-1)) = 1$.

Now we compute the decrypting exponent, d, which is the multiplicative inverse of e (mod $(p-1) \times (q-1)$). Use the extended gcd algorithm described in B.1.2. The following are the properties of e and d:

$$\begin{aligned} e \times d \ &\equiv\ 1 \ (mod\ (p-1)(q-1)) \\ (x^e)^d \ &\equiv\ x \ (mod\ n) \ \equiv\ (x^d)^e \ (mod\ n) \end{aligned}$$

Note that one can choose d first and calculate e, rather than the other way around.

A.2 Strong Primes

In the following, take "large" to mean more than 100 binary digits.

p^--strong

The largest prime factor of $p-1$, called p^-, must be large. Giving $p = a^-p^-+1$ for some a^- and large prime p^-. If p meats this requirement it is said to be p^--strong.

p^{--}-strong

If p^- itself is a strong prime, i.e. $p^- = a^{--}p^{--} + 1$, for some integer a^{--} and some large prime p^{--}, then we say that p is p^{--}-strong.

p^+-strong

If the largest prime factor of $p + 1$, called p^+, is large, giving $p = a^+p^+ - 1$, for some integer a^+ and large prime p^+, then p is said to be p^+-strong.

Strong

If p is p^--strong, p^{--}-strong, and p^+-strong, then we say that p is simply a strong prime.

Strong primes were first introduced by Rivest et al in the original paper. The idea was to make the state of the art factoring algorithms of the time ($p - 1$ and $p + 1$) work badly. However, the development of brand new factoring approaches such as the Elliptic Curve method, the Quadratic sieve and the Number Field Sieve and their improvements has left the strong prime condition obsolete. Using strong primes will give a negligible increase in security over simply using enormous primes. Randomly generated primes of 1024 bits are very likely to be impervious to $p \pm 1$ attacks (because $p \pm 1$ is unlikely to be smooth). However there is little extra computation required to generate strong primes. For the explanation, history of and algorithms to generate strong primes, please read Rivest [20] (available on the CD-ROM).

Appendix B

Tools

B.1 Greatest Common Divisor

Finding the gcd is done using the Euclidean Algorithm. Mathematica provides an implementation. The residue from the gcd calculations are often used to calculate multiplicative inverses for a particular modulus. I have also included a description of the extended gcd algorithm.

B.1.1 Euclidean Algorithm

To get the gcd of two integers a and b.

1. reorder so that $a > b$ in gcd(a, b)

2. write a in terms of its quotient and remainder: $a = b.q + r$

3. if $r = 0$, then return b as the gcd

4. go to step 1 with $a = b$ and $b = r$

An example is given in Figure B.1.

$$\begin{aligned}
& gcd(147, 45) \\
&= gcd(45, 12) \quad as \quad 147 = 45 * 3 + 12 \\
&= gcd(12, 9) \quad as \quad 45 = 12 * 3 + 9 \\
&= gcd(9, 3) \quad as \quad 12 = 9 * 1 + 3 \\
&= 3 \quad as \quad 9 = 3 * 3 + 0
\end{aligned}$$

Figure B.1: Example of the Euclidean algorithm

B.1.2 Extended Euclidean Algorithm

The extended algorithm returns g, the gcd(m, x), and two integers a, b such that

$$a \times m + b \times x = g$$

This is useful especially when trying to find the multiplicative inverse of x. We define the inverse to be x^{-1} so that $x \times x^{-1} \equiv 1\ (mod\ m)$. This can be written as

$$x^{-1} \times x + k \times m = 1$$

This looks like the output from the extended gcd. Note that a number only has a multiplicative inverse modulus another when they are coprime to one another (their gcd is equal to 1).

Let's go through the previous example to demonstrate. Since gcd(147, 45)=3, the system will not give a multiplicative inverse, but the procedure is important at this time.

Look at the second last line and rewrite it in terms of the remainder, 3:

$$3 = 12 - 9 \times 1 \tag{B.1}$$

Now look at the third last line. First rewrite it in terms of the remainder, 9, so that we have $9 = 45 - 12 \times 3$, and substitute it into B.1.

$$3 \quad = \quad 12 - (45 - 12 \times 3)$$

$$\begin{aligned} 3 &= 12 - 45 + 12 \times 3 \\ 3 &= 12 \times 4 - 45 \end{aligned} \tag{B.2}$$

Now take the first line of calculations, rewrite in terms of $12 = 147 - (45 \times 3)$ and substitute into B.2

$$\begin{aligned} 3 &= (147 - 45 \times 3) \times 4 - 45 \\ &= 4 \times 147 - 12 \times 45 - 45 \\ &= 4 \times 147 - 13 \times 45 \end{aligned} \tag{B.3}$$

And from this we can see that $a = 4$ and $b = -13$.

But it would be simpler to do the calculations in one pass rather than have to go back up the set of equations. I demonstrate this below. The tables show 5 columns, an index and four other labelled a, q, x and y. To start set up as in Table B.1.

i	a	q	x	y
1	147	–	1	0
2	45		0	1

Table B.1: The initial values of the table

a_1 and a_2 contain the two numbers we are getting the gcd of. x_1, x_2, y_1 and y_2 always have the values 1, 0, 0, 1 respectively. q_1 never has a value. We calculate the remaining places using the following rules.

- $q_i = \lfloor a_{i-1}/a_i \rfloor$ - i.e. the quotient
- $a_i = a_{i-2} - q_{i-1}a_{i-1}$ - i.e. the remainder
- $x_i = x_{i-2} - q_{i-1}x_{i-1}$
- $y_i = y_{i-2} - q_{i-1}y_{i-1}$

i	a	q	x	y
1	147	–	1	0
2	45	3	0	1
3	12	3	1	-3
4	9	1	-1	10
5	3	3	4	-13
	0			

Table B.2: The completed table for gcd(147,45)

The completed table is shown in Table B.2. Line 5 is where the information is found. The gcd is $a_5 = 3$ and the integers we wanted are in x_5 and y_5.

B.2 Fast Exponentiation

Mathematica provides a built in function, PowerMod, that performs fast exponentiation. This outperformed my own version only slightly, but I use the built in function throughout the programs. Here is an explanation of why it is so fast.

$$PowerMod[a,\ b,\ n] \ = \ a^b \ (mod\ n)$$

If the exponent is a power of 2 then we use successive squaring. Otherwise use the binary representation of b.

$$a^{32} \ = \ (((((a^2)^2)^2)^2)^2)$$

$$a^{37} \ = \ a^{32} \ \times \ a^4 \ \times \ a^1$$

Each squaring and multiplication is done mod n.

B.3 Primality Testing

B.3.1 Pseudoprimes

Theorem 4.1 stated that a prime could be identified by a simple test. If p is prime, then $2^{p-1} \equiv 1 \ (mod\ p)$. But this is not always the case. Consider $341 = 11 \times 31$, which gives $2^{340} \equiv 1 \ (mod\ 341)$ and therefore passes the test. Thus all primes must obey the law, but certain composite numbers also obey it. Therefore we call composite numbers that pass the test pseudoprimes.

The test is not limited to the base 2 however, we can use a generalised theorem

Theorem B.1 *If p is a prime that does not divide b, them*

$$b^{p-1} \equiv 1 \ (mod\ p)$$

Thus we have the basis for a test. Instead of testing under just one base, we try for many coprime bases. Our previous example fails the test under base 3: $3^{340} \equiv 56 \ (mod\ 341)$. Unfortunately there exist composite numbers, called Carmichael numbers, which are pseudoprimes for all coprime bases. An example of one is $561 = 3 \times 11 \times 17$. However they are extremely rare. Rivest presents empirical data [21] which suggests that the chances of a randomly chosen number that passes the pseudoprime test actually being composite is less than 1 in 10^{22}. But we are not finished there. If we need to be certain without doubt, there is another level of testing, the strong pseudoprime test.

B.3.2 Strong Pseudoprimes

However pseudoprimes are a little worrying. We should like to be able to avoid them. Take a number n which has passed the pseudoprime test for a certain base, b, therefore $b^{n-1} \equiv 1 \ (mod\ n)$. This is the same thing as saying that n divides $b^{n-1} - 1$. n must be odd (otherwise why test it for primality), so we can write it as $n = 2m + 1$.

n	pseudoprimes<n	strong pseudoprimes<n
10^3	3	0
10^6	245	46
10^9	5,597	1282
25×10^9	21,853	4,842

Table B.3: Comparison of pseudoprimes and strong pseudoprimes

This means that n divides

$$b^{2m} - 1 = (b^m)^2 - 1^2 = (b^m - 1)(b^m + 1)$$

by the difference of two squares. If n is prime and divides the left of the equation, then it must divide one of the factors on the right. It cannot divide both as then it should divide their difference but clearly $(b^m+1)-(b^m-1) = 2$. We can write the test to see if n divides one of the factors as

$$b^m \equiv 1 \text{ or } -1(mod\ n).$$

If n is a composite number, some of its factors will divide $b^m + 1$ and others will divide $b^m - 1$. It would pass the pseudoprime test for base b but it would not satisfy the above equation. Let us define a test for strong pseudoprimes based on this. Firstly we write n in terms of a and t:

$$n = 2^a \times t + 1$$

We can then write

$$b^{n-1} - 1 = (b^t - 1) \times (b^t + 1) \times (b^{2t} + 1) \times (b^{4t} + 1) \times \ldots \times (b^{2^{a-1}} + 1)$$

n is a strong pseudoprime if it divides exactly one of the factors on the right of this equation.

Table B.3, taken from Pomerance et al [17], shows the improvement of the strong-pseudoprime test over the normal one for the base 2.

The test can be improved by using a selection of bases. Note that there is no analogue to Carmichael numbers. No composite number can pass the test for all coprime bases. This gives us a good confidence level for the success of the test.

B.4 Chinese Remainder Theorem

Given integers $n_1, n_2, ..., n_k$ which are pairwise relatively prime, and a set of residues $a_1, a_2, ..., a_k$ given by

$$a_i \equiv x \ (mod\ n_i) \qquad i = 1 \ldots k$$

We can construct a system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \ (mod\ n_1) \\ x &\equiv a_2 \ (mod\ n_2) \\ &\vdots \\ x &\equiv a_k \ (mod\ n_k) \end{aligned}$$

The CRT states that there is a is a unique solution, x, less than $n = n_1 n_2 \ldots n_k$. The following algorithm is due to Guass and was taken from [15]

$$x = \sum_{i=1}^{k} a_i N_i M_i \ (mod\ n)$$

where $N_i = n/n_i$ and $M_i = N^{-1} \ (mod\ n_i)$. The Extended GCD (see above B.1.2) can be used to calculate N^{-1}.

Appendix C

Source Code