A Short Introduction to Brute-Forcing RSA

赖纳文

November 16 2020

Abstract

This paper, as the title suggests, introduces a brief overview on how to brute-force RSA cryptography. This could either mean given the encryption key and modulus, the decryption key would be retrieved, or given the decryption key and modulus, the encryption key would be retrieved.

A brief explanation of RSA and how to use it is also covered, although proof of its correctness is out-of-scope of this paper. [1]

1 Introduction

RSA is an assymmetric, or public-key cryptosystem, meaning that there are separate keys used for encryption and decryption. Details of the RSA cryptosystem were first publicly described in 1977 and a formal paper was published in 1978. [4] The basic premise is that the cryptosystem relies on the difficulty of factoring two prime numbers.

That is to say, where p and q are prime, and given:-

$$n = p \cdot q$$

It would be "difficult" (i.e: could not be done in polynomial time [2]) to find p and q given simply just n and e or d. [3] n could be factored in only one way using primes (excluding rearrangements), as proven by the *Fundamental Theorem of Arithmetic*. [4]

This process is called integer factorisation [5].

[1] For the curious, See section VI of the original RSA paper[4] which makes use *Fermat's Little Theorem*, or alternatively here: https://www.di-mgt.com.au/rsa_theory.html

[2] For now, I advise to just think of it [polynomial time] as " the time scaling depending on the value at a reasonable rate". If you are familiar with Big O notation, polynomial time is in Big O notation $O(n^c)$, don't worry too much if you don't understand.

[3] Technically speaking, we don't have mathematical proof it isn't possible (on classical computers at least, on Quantum computers there is an algorithm called Shor's Algorithm for this), it's an open unsolved problem in computer science

[4] This was proven by Euclid thousands of years ago and you can find a proof online.

[5] Or rather, *f*actorization for our American counterparts

2 Process

There are several processes involved, namely, the *encryption* process, *decryption* process, and the *key generation* process. We will be working backwards, starting with encryption/decryption first, so we can understand *how* RSA is used, then move on to how to generate RSA keys. First off, some key terms:-

2.0.1 n - *modulus*

The product of prime numbers p and q. This is given out along with the public key d. Sometimes when references are made to the "public key", it is implied that this modulus is included. See RFC8017 [3] Section A.1.1 for more information. Not to be confused with the 'Absolute Value' function of no relation also called Modulus.

2.0.2 e - *encryption key*

Also known as the public key. It is used to encrypt messages. Normally the number 65537 is used[6], although technically any number which is between 1 and the totient function (see next the totient function sections below) and is coprime[7] with the totient function can be used.

[6]Notice how 65537 is actually $2^{16}+1$, which is actually called a Fermat prime. It's used because it's faster to do certain calculations (one of the modular exponentiation algorithms) because this number in binary form is $10000.....1_2$ and only has two 1s.

[7]Also known as relatively prime or mutually prime. Basically an integer where the greatest common divisor of itself and the number it is coprime with is 1.

2.0.3 d - *decryption key*

Also known as the private key. It is used to decrypt messages.

2.0.4 C - *ciphertext*

The message in its encrypted form.

2.0.5 P - *plaintext*

The message before encryption, or equivalantly: the encrypted message after being decrypted.

2.0.6 ϕ - *Euler's totient function*

This was the totient function used in the original RSA paper.[4] The result of $\phi(n)$ is the number of positive integers less than n that are coprime with n itself.

Euler's product formula states

$$\phi(n) = n\prod_{p|n}(1-\frac{1}{p})$$

But there is a short way to do this in our case, given $n = p \cdot q$ where p and q are primes, $\phi(n)$ can be derived from p and q like so[8] :-

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

2.0.7 λ - *Carmichael's totient function*

This is an improved 'reduced' totient function that is used in place of Euler's totient function sometimes. It's used in the PKCS#1 RFC8017[3], which is a

[8]See: http://www.math.unl.edu/ tmarley1/math189/notes/Nov20notes.pdf for proof

specification for an RSA implementation. It gives a different and smaller number than Euler's totient function, while still functioning the same in context of RSA. $\lambda(n)$ can be derived from primes p and q like so:-

$$\lambda = lcm((p-1),(q-1))$$

where lcm is the lowest common multiple.[9]

Now, we can move on to the actual processes

2.1 Encryption

The ciphertext C can be derived from the plaintext, encryption key, and modulus like so:-

$$C \equiv P^e \mod n$$

2.2 Decryption

The plaintext P can be derived from the ciphertext, decryption key, and modulus like so:-

$$P \equiv C^d \mod n$$

You may notice that since it is $\mod n$, the resulting plaintext can must be less than or equal to the modulus ($P \leqslant n$), and yes that is indeed true. For that reason we can only encrypt messages which in length are less than or equal to the modulus. In practice, very large numbers for p and q are used and this isn't a problem. [10]

2.3 Key Generation

First, compute the modulus n as the product of two arbritatrily chosen primes. [11]

$$n = p \cdot q$$

Then we need to find d and e such that:-

$$d \cdot e \equiv 1 \mod \phi(n)$$

or alternatively

$$d \cdot e \equiv 1 \mod \lambda(n)$$

That is to say, d and e are modular multiplicative inverses of each other $\mod \phi(n)$ or $\lambda(n)$.

You then choose either the decryption key d or the encryption key e by being

[9] There exists a formula for calculating the lcm using the gcd (which you can, of course, find using the Euclidean algorithm), $lcm(a,b) = \frac{|a \cdot b|}{gcd(a,b)}$. Proof not provided.

[10] Furthermore, since RSA is 'slow', usually a separate faster symmetric encryption system is used, and RSA is simply used to encrypt the key for that cryptosystem, which need not be so long.

[11] If the purpose was for stronger encryption, they won't be so arbritarily chosen. Generally, bigger numbers are better as it will take longer to break. The original paper [4] recommends around 100 digits for p and q to end up with a roughly 200 digit modulus n. It also recommends for the two numbers to differ in length by a few digits to protect against "sophisticated factoring algorithms" which is not covered in this paper. For the curious, this is through Fermat Factorisation, which you are likely familiar with from Secondary School, even if you didn't know the name of it. It is difference of two squares. $a^2 - b^2 = (a+b) \cdot (a-b)$

choosing an integer which is coprime to a totient function (either one. Euler's totient function was used in the original RSA paper, Carmichael's totient function is the 'improved' version), see above.

Note that some sources may say something like 'determine $d = e^{-1} \equiv \phi(n)$ ' which is an abuse of notation since you cannot solve that since e^{-1} is not an integer value.

Instead, solve it using the Extended Euclidean Algorithm like a normal linear congruence. For example, for an linear congruence $3x \equiv 1 \mod 5$, change it into the form $3x - 5k = 1$ where k is an integer, and find the greatest common divisor using the Euclidean Algorithm of 3 and 5, then use the Extended Euclidean Algorithm to find an integer value for x. Do recall that there are multiple integer solutions for x and k (Refer to lecture notes and recorded lectures and/or tutorials), usually we will use a positive value for x since that is easier to work with.

2.4 Fast Modular Exponentiation *(extra)*

Not strictly neccessary for small numbers, but for a large number you may be able to optimise modular exponentiaion (used in encryption/decryption) by splitting the mod for each variable. For example, an example using m is 5, the modulus is 22, and e is 13.[12]

$$
\begin{aligned}
C &\equiv m^e \mod n \\
C &\equiv 5^{13} \mod 22 \\
C &\equiv (5^2)^3(5) \mod 22 \\
C &\equiv (25 \mod 22)^3(5) \mod 22 \\
C &\equiv (3)^6(5) \mod 22 \\
C &\equiv (729 \mod 22) * 5 \mod 22 \\
C &\equiv 15 \mod 22
\end{aligned}
$$

Thus we don't have to compute the large number 5^{13} and can simply work with smaller numbers like 3^6 and 5, which is especially useful if you are doing 64-bit calculations and resulting numbers of m^e are larger than that. [13]

Details of implementation in code is left as an exercise to the reader.[14]

3 Brute-Forcing to find p and q

Finally on to the most interesting part. From subsection **2.3** we can see an equation involving d, e and n. Great! We have e and n, it should be trivial to get d right, by way of Euler's Extended Algorithm?

[12] This example was taken directly from one of the MAT1830 Lectures.

[13] Python, as always, already has a function that does modular exponentiation, and in any case, numbers in python aren't restricted to any number of bits. But I thought this was an interesting thing to discuss, and it would come in handy if you were working in a lower-level language.

[14] Some key terms which may be useful are 'modular exponenetiation' or 'modpow', with the 'pow' standing for 'power'.

Not exactly. We need to calculate $\phi(n)$ or $\lambda(n)$ to use as the modulus. Previously we used a short way by using either Euler's totient function or Carmichael's totient functions; but that only works if we know the values of p and q!

For now, let's assume the key generation is using Euler's totient function. If you don't recall, Euler's totient function returns the number of integers less than n that are coprime with n itself. And so we start with this basic algorithm to find $\phi(n)$

Input: modulus n
Output: Euler's totient function (number of integers $< n$ that is coprime to n)
set counter to 0;
for *numbers in range 1 through n inclusive* **do**
  if *gcd(current_number, n) is equal to 1* **then**
    add 1 to counter;
  end
end
return counter;

Algorithm 1: Euler's Totient Brute Force

We could also just manually use the product sum rule as mentioned above. Also note that it's assuming we're using Euler's totient function, so it won't work if Carmichael's totient function was used. Alternatively, since we know that $\phi(n)$ can be computed using $(p-1)(q-1)$ where p and q are prime numbers, and $p \cdot q = n$, and we have n, and we know that there is only one way to factorise n[15], we can instead opt to find p and q, so that we can compute $\phi(n)$ easily using the short method of $(p-1) \cdot (q-1)$

Input: modulus n
Output: The prime factors of n
for *numbers in range 2 through n inclusive* **do**
  if *current number is divisible by n* **then**
    return current number and $\frac{n}{currentnumber}$;
  end
end

Algorithm 2: Prime Factorisation Brute Force

Since this method retrieves both p and q, it could work with either totient function.

There are a number of optimisations that can be done. For example, we only need to loop through until $\lfloor\sqrt{n}\rfloor$ (If you're not familiar with the notation of floor, it is simply rounding down.) [16]. And since we know that it can be computed using $(p-1)(q-1)$ where p and q are prime numbers, we can get a list of prime numbers, and try to see if modulus n divides a certain prime [17] minus 1, for example $r-1$ where r is a prime number. The trade-off of course being that you would have to maintain a list of prime numbers.

As a reminder, once p and q are obtained, one could go through the key

[15] As I mentioned earlier, through way of the Fundamental Thoerem of Arithmetic

[16] The proof isn't long, try it yourself by using a proof by contradiction. Also perhaps starting from $\sqrt{n}$ and then working backwards might be faster.

[17] If you recall, you can check if a number divides another by checking if the remainder is 0 during division

generation step as in subsection **2.3**, and then use the Extended Euclidean Algorithm to obtain d.

4 Further Reading

At the time of writing, the wikipedia article on RSA[5] is fairly comprehensive if you want a deeper understanding of RSA. For further information on attacking RSA, there is this paper *"Twenty Years of Attacks on the RSA Cryptosystem"* [1] which covers methods such as "Coppersmith's Attack" and "Wiener's Attack". There is also *Revisiting Fermat's Factorization for the RSA Modulus* [2] which covers Fermat's Factorisation for finding p and q.

References

[1] Dan Boneh. Twenty years of attacks on the rsa cryptosystem. *NOTICES OF THE AMS*, 46, 02 2002.

[2] Sounak Gupta and Goutam Paul. Revisiting fermat's factorization for the rsa modulus, 2009.

[3] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. Pkcs #1: Rsa cryptography specifications version 2.2. RFC 8017, November 2016.

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

[5] Wikipedia. RSA (cryptosystem) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=RSA%20(cryptosystem)&oldid=989291810`, 2020. [Online; accessed 20-November-2020].