

Decrypting Attacks on RSA

Yael Zayats

July 16, 2023

Abstract

In this paper, we explore a few methods of attacking the RSA cryptosystem. These attacks are categorized into 3 different sections: guess and check, faults and human error, and factorization methods. Commentary is added on the effectiveness of these attacks, fixes for security, and possible ways to improve in the future.

1 Background

The RSA cryptosystem was invented by Ron Rivest, Adi Shamir, and Len Adleman at MIT in 1977. It is widely used in numerous domains, ranging from secure communication systems to digital signatures and online transactions.

At its core, RSA operates as a public-key encryption method, meaning it employs a pair of keys: a public key and a private key. The public key is openly shared and readily accessible to anyone, while the private key is securely held by each individual user.

The public key is made up of 2 distinct values, N and e , where:

$$N = p \cdot q, \text{ and}$$

$$e = (p - 1) \cdot (q - 1) .$$

p and q are both prime numbers.

The private key is the value d , where:

$$d = e^{-1} \pmod{(p - 1)(q - 1)} = e^{-1} \pmod{\varphi(N)}$$

Where $\varphi(N)$ is Euler's totient function, that is, there exists an integer k such that $ed - k\varphi(N) = 1$. An easier way to think about this is $\varphi(N) = (p - 1)(q - 1)$.

To encrypt a message M , where $M < N$ and $M \in \mathbb{Z}_N^*$, into a cipher text C , the user will use the equation:

$$C = M^e \pmod{n}$$

Conversely, to decrypt a message C into a cipher text M , the user will use the equation:

$$M = C^d \cdot \pmod{n}$$

The key length of RSA is the length of the modulus n in bits. Currently, the minimum recommended length is at least 1024 bits. Even if it's not trivial to break for the common person, a key with a bit length of 512 is no longer considered secure. There are a few conventions associated with the key length. One convention is that the key length is the position of the most significant bit in n that has value '1', where the least significant bit is at position 1. The most significant byte 0x0A in binary is 00001010'B. The most significant bit is at position 508, so its key length is 508 bits. On the other hand, this value needs 64 bytes to store it, so the key length could also be referred to by some as $64 \times 8 = 512$ bits. Equivalently, key length = $\lceil \log_2(n + 1) \rceil$. The other convention is that the key length is the number of bytes needed to store n multiplied by 8, $\lceil \log_{256}(n + 1) \rceil \cdot 8$. In practice, nothing shorter than 1024 bits is used.

One example is the key $N = 119294134840169509055527211331255649644606569661527638012067481954943056851150333806315957037715620297305000118628770846689969112892212245457118060574995989517080042105263427376322274266393116193517839570773505632231596681121927337473973220312512599061231322250945506260066557538238517575390621262940383913963$, which is composed of the

primes $p = 10933766183632575817611517034730668287155799984632223454138745671121273456287670$
 $008290843302875521274970245314593222946129064538358581018615539828479146469$ and $q = 1091061$
 $6967349110231723734078614922645337060882141748968209834225138976011179993394299810159736904$
 $468554021708289824396553412180514827996444845438176099727$. Scary!

The following table is taken from NIST's Recommendation for Key Management [NIST-80057] [Bar20]. It shows the recommended comparable key sizes for symmetrical block ciphers (AES and Triple DES) and the RSA algorithm. That is, the key length you would need to use to have comparable security.

Symmetric Key Algorithm	Comparable RSA key length	Comparable hash function	bits of security
2TDEA	1024	SHA-1	80
3TDEA	2048	SHA-224	112
AES-128	3072	SHA-256	128
AES-192	7680	SHA-384	192
AES-256	15360	SHA-512	256

The strength of the algorithm comes from these one way functions - they are easy to calculate but extremely difficult to reverse. By withholding the prime numbers used in the key generation process confidential, it becomes challenging, though not impossible, for an unauthorized party to deduce the necessary components for decryption. This paper introduces numerous methods of RSA decryption, attacking not only the cipher itself but the systems using it.

2 Guess and Check Methods

2.1 Searching the Message Space

Public key cryptography has a weakness in the fact that part of it must be given away. With a small enough message space, an attacker can simply try to encrypt every message block, until a match is found with the corresponding section of ciphertext. In practice, this would only work with tiny message spaces, and would take too long to be a realistic approach to decrypt any actual usage of RSA.

2.2 Guessing d

This is also called a "known ciphertext attack". This attack requires the attacker to know both the plaintext and the ciphertext. They then try to guess the key to discover the private key d . Once d has been found, it is easy to find the factors of N , at which point the system has been cracked. From here, all ciphertexts can be decrypted.

The issue with this attack is that it is very slow, as trying every d can be very time consuming.

2.3 Cycle Attack

Another potential attack on public key cryptography involves repeatedly encrypting the ciphertext until the original plaintext is obtained, counting the number of iterations required. Essentially, you calculate $M^e \pmod{N}$, $M^{e^2} \pmod{N}$, ... and so on until, for some k , $M^{e^k} \pmod{N} = M$.

This method, although theoretically possible, is exceedingly slow and impractical, particularly when dealing with large encryption keys. A more generalized version of this attack leverages the factorization of the modulus and can be faster in most cases. However, even with this approach, the difficulty significantly increases when a large key is used.

2.4 Wiener's Theorem

The theorem states that if the private exponent in RSA is smaller than a certain threshold related to the size of the modulus, an attacker can efficiently recover the private key and decrypt encrypted messages.

Theorem Let $N = pq$ with $p < q < 2q$. Let $d < \frac{1}{3}N^{\frac{1}{4}}$. Given (N, e) with $ed \equiv 1 \pmod{\varphi(N)}$, d can be efficiently recovered by searching the right $\frac{k}{d}$ among the convergents of $\frac{e}{N}$.

This attack is based on the continued fraction representation of the private exponent and its relation to the convergents. By analyzing these convergents, an attacker can determine the private key with high probability. We may start by noting the congruence $ed \equiv 1 \pmod{\varphi(n)}$ can be written as the equality $ed = k\varphi(n) + 1$ for some value k . We can additionally note that $\varphi(n) = (p-1)(q-1) = pq - p - q + 1$, since both p and q are much shorter than $pq = n$, we can say that $\varphi(n) \approx n$. Dividing the former equation by $d\varphi(n)$ gives us $\frac{e}{\varphi(n)} = \frac{k+1}{d}$, and using the approximation, we can write this as $\frac{e}{n} \approx \frac{k}{d}$. An important part of this theorem is that the left-hand side of the equation is made up of entirely public information.

It is possible to factor n by knowing n and $\varphi(n)$. Consider the quadratic polynomial $(x-p)(x-q)$, which has the roots p and q . Expanding it gives us the polynomial $x^2 - (p+q)x + pq$, and by substituting for the variables we can write this as $x^2 - (n - \varphi(n) + 1)x + n$. Applying the quadratic formula gives us both p and q , where $a = 1$, $b = n - \varphi(n) + 1$, and $c = n$.

This attack works by expanding $\frac{e}{n}$ to a continued fraction and iterating through the terms to check various approximations $\frac{k}{d}$. In order to make this checking process more efficient, we can make a few observations:

- since $\varphi(n)$ is even, and e and d are both by definition coprime to $\varphi(n)$, we know that d is odd
- Given the above equations and the values of e, n, d , and k , we can solve for $\varphi(n)$ with the equation $\varphi(n) = \frac{ed-1}{k}$, thus we know that $ed-1$ has to be divisible by k
- if the $\varphi(n)$ is correct, the polynomial $x^2 - (n - \varphi(n) + 1)x + n$ will have roots p and q , which we can verify by checking that $pq = n$

3 Faulty Encryption & Human Error

3.1 Common Modulus

In this case, users within an organization have shared the public modulus, such as an administration choosing a secure public modulus and generating keys for its employees and users. This is very convenient for software writing and management, but it can give away information to various groups. If the same N is used by all users, but a central authority provides each user i with a unique pair of e_i and d_i , each user now has a "unique" set of keys. While at first glance this may seem to work, since any ciphertext C intended for one user cannot be decrypted by another as they do not have the same d_i , this notion is incorrect. Eve can use her own exponents e_e, d_e to factor the modulus N . Once N is factored, she can recover any other private key from the known public key of the user. This is easily fixed by not sharing a modulus between multiple users.

Another example would be that any message encrypted with 2 keys can be broken by an eavesdropper, such as a memo sent to multiple employees [SIM83].

Another issue is that this sort of system is at risk of decryption from insiders, who could break the whole system and view anybody's messages [DEL84].

3.2 Faulty Communication

In a situation where an attacker has access to a user communication channel, as in, the attacker can listen to and change what is being transmitted, the encryption system is at risk of being broken. Let's call the users Alice and Bob, and the attacker Eve.

If Alice decides to talk to Bob, but does not know his public key, she can request it over the faulty communication channel. Bob replies over the same channel. However, during the communication, Eve intercepts the message and sees the public key. She flips a single bit in Bob's public key, changing it from (e, N) to (e', N) .

When Alice receives the faulty public key, she uses it to encrypt her message and sends it to Bob. Bob cannot decrypt it since the wrong key was used, and tells Alice to try again. Once again, he sends his public key. This time Eve does not interfere and Alice is able to encrypt and send her message.

Now, assuming that Alice encrypted the same text twice, Eve has two ciphertexts, as well as the full public keys. Now, she can use the public modulus attack in order to retrieve Alice's message [JQ87].

This attack only works with several faults in communication, both within the channel and the human decision to encrypt the same message twice. With a change in either, this attack would not work.

3.3 Low Public Exponent (e)

To reduce calculation time, a small public exponent is used, often at a minimum of $e = 3$. It is recommended to use $e = 2^{16} + 1$, as it only requires 17 multiplications for signature verification and is large enough to deter most low e attacks.

3.3.1 Coppersmith Theorem


Theorem Let N be an integer and $f \in \mathbb{Z}[x]$ be a monic polynomial of degree y . Set $X = N^{1/d-\epsilon}$ for some $\epsilon \geq 0$. Then, given (N, f) , an attacker can find all integers $|x_0| < X$ satisfying $f(x_0) \equiv 0 \pmod{N}$. [Cop97]

The theorem provides an algorithm for efficiently finding all roots of $f \pmod{N}$ that are less than $X = N^{\frac{1}{d}}$. The algorithm's running time decreases as X gets smaller. The strength of this theorem is its ability to find small roots of polynomials modulo a composite N . Given a polynomial $h(x) = \sum a_i x^i \in \mathbb{Z}[x]$, define $\|h\|^2 = \sum |a_i|^2$. The proof relies on the following observation:

Lemma Let $h(x) \in \mathbb{Z}[x]$ be a polynomial of degree d and let X be a positive integer. Suppose $\|h(xX)\| < N/\sqrt{d}$. If $|x_0| < X$ satisfies $h(x_0) \equiv 0 \pmod{N}$, then $h(x_0) = 0$ holds over the integers [How97].

Proof Observe from the Schwarz inequality that

$$\begin{aligned} |h(x_0)| &= |\sum a_i x_0^i| = |\sum a_i X^i (\frac{x_0}{X})^i| \\ &\leq \sum |a_i X^i (\frac{x_0}{X})^i| \leq \sum |a_i X^i| \leq \sqrt{d} \|h(xX)\| < N \end{aligned}$$

Since $h(x_0) \equiv 0 \pmod{N}$, we conclude that $h(x_0) = 0$. 

The lemma states that if h is a polynomial with low norm, then all small roots of $h \pmod{N}$ are also roots of h over the integers. The lemma suggests that to find a small root x_0 of $f(x) \pmod{N}$ we should look for another polynomial $h \in \mathbb{Z}[x]$ with small norm having the same roots as $f \pmod{N}$. Then x_0 will be a root of h over the integers and can be easily found. Coppersmith found a trick to solve the problem: if $f(x_0) \equiv 0 \pmod{N}$, then $f(x_0)k \equiv 0 \pmod{N^k}$ for any k . More generally, define the following polynomials:

$$g_{u,v}(x) = N^{m-v} x^u f(x)^v$$

for some predefined m . Then x_0 is a root of $g_{u,v}(x) \pmod{N^m}$ for any for any $u \geq 0$ and $0 \leq v \leq m$. To use the previous Lemma, we must find an integer linear combination $h(x)$ of the polynomials $g_{u,v}(x)$ such that $h(xX)$ has norm less than N^m . Thanks to this upper bound on the norm, we can show that for a sufficiently large m there always exists a linear combination $h(x)$ satisfying the required bound. Once $h(x)$ is found, the lemma implies that it has x_0 as a root over the integers. Consequently x_0 can be found.

The efficiency of this theorem can be proved using the LLL Algorithm

This theorem can be used to attack stereotyped messages (messages whose difference is less than $N^{\frac{1}{e}}$), polynomially related messages (identical messages sent to multiple recipients), and factoring N if the high bits of p are known.

3.3.2 Hastad's Broadcast Attack


Suppose Bob sends a message to a number of parties, P_1, P_2, \dots, P_k . Each party has its own public key, (e_i, N_i) , where $M < N$. In this case, Bob would encrypt each party's message with its own public key, and send a separate encrypted message to each one. Eve can then eavesdrop and collect the k sent messages.

In this case, Eve can recover the message if $k \geq e$, where $C_n = M^k \pmod{N}$. We may assume that $\gcd(N_i, N_j) = 1$ for all $i \neq j$ since otherwise Eve can factor some of the N_i 's. Hence, applying the Chinese Remainder Theorem to C_1, \dots, C_k gives a $C \in \mathbb{Z}_{N_1 \dots N_k}$ Satisfying $C' = M^k \pmod{N_1, \dots, N_k}$. Since M is less than all of the N_i 's, we have $M^k < N_1 \dots N_k$. Then $C' = M^k$ holds over the integers. Thus, Eve can recover M by computing the real cube root of C' . More generally, if all public exponents are equal to e , Eve can recover M as soon as $k \leq e$.

3.3.3 Franklin-Reiter Related Message Attack

If two related messages, as in, two messages with a fixed difference, $M_1 = f(M_2) \pmod{N}$, are encrypted with the same modulus, it is possible to decrypt it under any e .

Lemma Set $e = 3$ and let (N, e) be an RSA public key. Let $M_1! = M_2$ satisfy $M_1 = f(M_2) \pmod{N}$ for some linear polynomial $f = ax + b$ with $b! = 0$. Then, given (N, e, C_1, C_2, f) an attacker can recover M_1 and M_2 in time quadratic in $\log(n)$.

Proof Since $C_1 = M_1^e \pmod{N}$, we know that M_2 is a root of the polynomial $g_1(x) = f(x)^e - C_1$ and similarly M_2 is a root of $g_2(x) = f(x)^e - C_2$. The linear factor $x - M_2$ divides both polynomials. Therefore, an attacker can use the euclidean algorithm to compute the gcd of g_1 and g_2 . If the gcd turns out to be linear, M_2 is found. 


We show that when $e = 3$ the gcd must be linear. The polynomial $x^3 - C_2$ factors modulo both p and q into a linear factor and an irreducible quadratic factor. Since g_2 cannot divide g_1 , the gcd must be linear. For $e > 3$ the gcd is almost always linear. However, for some rare $M_1; M_2$, and f , it is possible to obtain a nonlinear gcd, in which case the attack fails. For $e > 3$ the attack takes time quadratic in e . Consequently, it can be applied only when a small public exponent e is used. For large e the work in computing the gcd is prohibitive.

3.3.4 Coppersmith's Short Pad Attack

When sending a message, padding is added around it to increase randomness and ensure byte formatting. However, if used improperly, it allows an attacker access to the message.

Suppose Bob sends Alice a padded message. Eve intercepts it, and stops it from reaching Alice. Bob resends the same message, this time with random padding. Eve now has 2 messages, and is able to decrypt them despite not knowing the padding. [Hås88]

Theorem Let (N, e) be a public key where N is n bits long. Set $m = \lfloor n/e^2 \rfloor$. Let $M \in \mathbb{Z}/N\mathbb{Z}^*$ be a message of length at most $n - m$ bits. Define $M_1 = 2^m M + r_1$ and $M_2 = 2^m M + r_2$, where r_1 and r_2 are distinct integers with $0 \leq r_1, r_2 < 2^m$. If Eve is given (N, e) and the encryptions C_1, C_2 of M_1, M_2 (but is not given r_1 or r_2), she can efficiently recover M .

Proof Define $g_1(x, y) = x^e - C_1$ and $g_2(x, y) = x^e - C_2$. We know that when $y = r_2 - r_1$, these polynomials have M_1 as a common root. In other words, $\Delta = r_2 - r_1$ is a root of the resultant $h(y) = \text{res}_x(g_1, g_2) \in \mathbb{Z}_N[y]$. The degree of h is at most e^2 . Furthermore, $|\Delta| < 2^m < N^{\frac{1}{e^2}}$. Hence, Δ is a small root of $h \pmod{N}$, and Eve can efficiently find it using Coppersmith's Theorem. Once Δ is known, the FR attack from the previous section can be used to recover M_2 and M . 

3.3.5 Partial Key Exposure Attack

This attack is possible when the public key is small. If an attacker exposed a fraction of the bits of d , he can, on the assumption that the modulus is small, reconstruct the rest of d . [BDF98]

Theorem 3.3.5.1 Let (N, d) be a private key with N being n bits long. Given the $\lceil \frac{n}{4} \rceil$ least significant bits of d , an attacker can reconstruct all of d in time linear in $e \log_2 e$.

Theorem 3.3.5.2 (Coppersmith) Let $N = pq$. Given the $\frac{n}{4}$ least or the most significant bits of p , one can factor N efficiently. k integer exists such that $ed - k(N - p - q + 1) = 1$.

Since $d < \varphi(N)$, then $0 < k \leq e$. Reducing N to $2^{n/4}$ and setting $q = N/p$, we get $(ed)p - kp(N - p + 1) + kN = p(\text{mod } 2^{n/4})$

Since Eve is given the $n/4$ least significant bits of d , she knows the value of $ed \pmod{2^{n/4}}$. Consequently, she obtains an equation in k and p . For each of the e possible values of k , Eve solves the quadratic equation in p and obtains a number of candidate values for $p \pmod{2^{n/4}}$. For each of these candidate values, she runs the algorithm of Theorem 3.3.5.2 to attempt to factor N . One can show that the total number of candidate values for $p \pmod{2^{n/4}}$ is at most $e \log_2 e$. Hence after at most $e \log_2 e$ attempts, N will be factored.

3.4 Blinding

One common use of RSA is to attach encrypted signatures. Eve can ask Bob to sign a random "blinded message", one where he does not know what the message is. Most signature systems attach a one-way hash, and since it is fairly secure, Bob's system would attach a valid signature.

RSA does not inherently have a system to check whether a message is dangerous or not, so most systems will look for specific characters or strings to determine if it should sign the message. This is easily bypassed, such as by multiplying the dangerous message with a prime number.

To apply the attack, Eve simply has to choose a buffer r^e , where r is any small integer. She then sends her message with the applied buffer, $M' = (r^e \cdot M) \text{mod}(N)$. Since this now looks like a string of random characters rather than the searchable dangerous message, Bob's system signs the message, $S' = M'^d \cdot \text{mod}(N)$, and sends it back. To take out the blinding factor, Eve simply has to calculate $(\frac{S'^e}{r^e}) \cdot \text{mod}(N)$.

3.5 Timing Attack

Timing attacks exploit the fact that cryptographic operations can take varying amounts of time depending on the input and private keys. By measuring the execution time of RSA private key operations, it is possible to analyze the timing patterns and potentially uncover the secret key used in the calculations.

Let us consider a simple example of implementing a string compare function, where we have to return true if the strings match and return false if they don't.

```
def string\_compare(s1,s2):
    if len(s1) != len(s2):
        return False
    for i in range(len(s1)):
        if s1[i] != s2[i]:
            return False
    return True
```

In the above function, we first compare the two strings' lengths and then compare the individual characters if they are of equal length. While iterating over the loop, if we encounter two different characters, we return false. If we do not find any difference, we return true. Let us assume that the correct password is 5263987149, and the attacker starts guessing from the first digit (beginning from 0000000000). He measures that the system returns false after x seconds. After getting the same response time x for the first five guesses (i.e., from 0000000000 to 4000000000), he notices that the system takes a slightly longer time to respond ($x + \Delta x$) when he tries 5000000000. This is because the for loop during its first iteration doesn't return false since the first digit of the user input and the stored password is equal. Hence, the loop runs another iteration, thereby taking more time (Δx). Therefore, the attacker knows that the first digit he tried now is correct. He can repeat the same procedure to guess the remaining digits by observing the pattern of Δx . In this way, it would take the

attacker only $10 \cdot 10$ guesses at the maximum to find the correct password, compared to 10^{10} possible combinations while trying to brute force it.

For this attack, the attacker must have the target system compute $C^d \pmod{N}$ for a couple selected values of C . Then, the attacker measures the time required to perform computations, which allows them to recover the private key d bit by bit. The sample size required is proportional to the number of bits in the private key, and since the number of bits is finite, this calculation is computationally practical.

How to use:

1. Let $d = d_n d_{n-1} \dots d_0$ (binary of d)
2. Set $z = M$ and $C = 1$. For $i = 0 \dots n$ do:
 - (a) if $d_i = 1$ set $C = C \cdot z \pmod{N}$
 - (b) $z = z * z \pmod{N}$
3. C at the end has the value $M^d \pmod{N}$

3.6 Bleichenbacher's Attack on PKCS 1

Bleichenbacher's Attack on PKCS 1 (also known as the RSA padding oracle attack) is a cryptographic vulnerability that targets the RSA encryption scheme using PKCS 1 padding. This attack exploits the behavior of servers that provide different error responses based on the validity of the padding in RSA ciphertexts. By interacting with such a server and carefully analyzing the server's responses, an attacker can gradually obtain information about the private key.

The attack relies on a mathematical property of RSA decryption that allows an attacker to iteratively narrow down the possible value of the decrypted message. By repeating this process multiple times, an attacker can eventually deduce the complete plaintext message.

Let N be an n -bit RSA modulus and M be an m bit message with $m < n$. Before applying RSA encryption it is natural to pad the message M to n bits by appending random bits to it. An old version of a standard known as Public Key Cryptography Standard 1 (PKCS 1) uses this approach. After padding, the message is n bits long, starting with a 16-bit long block containing "02", followed by a random pad, followed by "00" and finally the message M . The "02" is added to signal that a random pad has been attached to the message. The 16-bit block does not have to be "02", it can be any specified block.

When a PKCS 1 message is received by Bob's machine, an application (such as a web browser) decrypts it, checks the initial block, and strips off the random pad. However, some applications check for the "02" initial block and if it is not present they send back an error message saying "invalid ciphertext". Bleichenbacher showed that using this error message, an attacker can decrypt ciphertexts of his choice.

Suppose Eve intercepts a ciphertext C intended for Bob and wants to decrypt it. Eve picks a random $r \in \mathbb{Z}_N^*$, computes $C' = rC \pmod{N}$, and sends C' to Bob's machine. Bob's checking application receives the new message and attempts to decrypt it. His machine responds with either an error message or does not respond at all, depending on if C' was properly formatted, aka has the correct initial 16-bit block. From this response, Eve can tell if she has guessed the initial block correctly, or can make another guess and send the message again. Bleichenbacher showed that this is sufficient for decrypting C . [Ble98]

3.7 Random Faults

Very rarely, a key can be chosen that encrypts a message to itself. This would allow the attacker access without any work on their part. However, these results are far more theoretical, and their randomness makes them impractical to use on a larger scale.

In addition, our computers can encounter bugs and make mistakes within calculations. If a glitch causes someone's computer to miscalculate a single instruction, such as flipping a bit, an attacker can easily factor N using the invalid signature. This sort of glitch was present in an early version of the Pentium chip, and can be caused by a hardware bug or ambient EM interference [BDF98].

A version of this error can occur while Bob is generating a signature. Suppose a single error occurs, and as a result, exactly one text letter C_p or C_q will be computed incorrectly. Say C_p is correct, but \hat{C}_q is not. The resulting signature is now $\hat{C} = T_1C_p + T_2\hat{C}_q$. Once Eve receives \hat{C} , she knows that it is a false signature since $\hat{C}^e \neq M \pmod{N}$. However, notice that $\hat{C} = M \pmod{p}$ while $\hat{C}^e \neq M \pmod{q}$. As a result, $\gcd(N, \hat{C}^e - M)$ exposes a nontrivial factor of N .

For the attack to work, Eve must have full knowledge of M . Namely, we are assuming Bob does not use any random padding procedure. Random padding prior to signing defeats the attack. A simpler defense is for Bob to check the generated signature before sending it out to the world. Checking is especially important when using a Chinese Remainder Theorem speedup method, such as the ones used by many corporate implementations.

4 Factorization Methods

4.1 Fermat's Factorization

This method of factorization consists of finding an x and y such that $x^2 - y^2 = z$. The right side of the equation factors into $(x - y)(x + y)$, and if $x - y$ is not one, then you have found a non-trivial factorization.

Fermat's method is not very efficient in finding factors themselves, but are theoretically important in that many more modern methods such as: the quadratic sieve, multiple polynomial quadratic sieve, and the special and the general number field sieves are all based upon this method.

4.2 Pollard's p-1 Algorithm

The $p - 1$ algorithm was developed by J.M.Pollard in the 1970's . The basic idea of the algorithm is to use some information about the order of an element of the group \mathbb{Z}_p to find a factor p of N [Cha05]. The algorithm is based on Fermat's little theorem:

Theorem 4.2: *Fermat's Little Theorem* Let p be a prime and $a \in \mathbb{Z}$ such that $p \nmid a$. Then, $a^{p-1} \equiv 1 \pmod{p}$

Proof Consider the following two sets of equivalence classes:

$$A = \{[a], [2a], [3a], [4a], \dots, [(p-1)a]\}$$

$$B = \{[1], [2], [3], [4], \dots, [(p-1)]\} = \mathbb{Z}_p - \{[0]\}$$

$A \subseteq B$ since $p \nmid a$ and p divides none of $1, 2, 3, \dots, p-1$.

Suppose that $\exists r, s \in \mathbb{N}$ such that $1 \leq r \leq s \leq p-1$ and $[ra] = [sa]$.


Then, $r - s = 0$ since $[a] \neq \text{mod}(p)$ (since $p \nmid a$)

Since $r \leq p$ and $s \leq p$, $r - s \equiv 0 \pmod{p} \Rightarrow r = s$

So, $[a], [2a], [3a], \dots, [(p-1)a]$ are all different \pmod{p} , which means that the cardinality of A is $p-1$. And, since $\#A = p-1, \#B = p-1$ and $A \subseteq B$, we can conclude that $A = B$, that is the equivalence classes in A are congruent to the equivalence classes of B under a certain rearrangement.

Hence, $a * 2a * 3a * \dots * (p-1)a = 1 * 2 * 3 * \dots * p-1 \pmod{p}$ $a^{p-1} * (p-1)! = (p-1)!$

$a^{p-1} = 1$, since $(p-1)! \neq 0 \pmod{p}$

This shows that $A = B$ 

How to use the algorithm:

1. Choose a bound for B for the algorithm, usually $10^5 - 10^6$

2. Compute $m = \prod_{\substack{p \text{ prime} \\ 1 \leq p \leq B}} p^{\lfloor \log B / \log p \rfloor}$

3. choose a random positive integer a between 1 and n
4. compute $d = \gcd(a, n)$
 - (a) if $d = 1$ go to 5
 - (b) if $d \neq 1$ return d (it is a non-trivial factor of n)
5. compute $a^m \pmod{n}$
6. compute $e = \gcd(a^m - 1, n)$
 - (a) if $e = 1$ go to 1 and increase B
 - (b) if $e = n$ go to 3 and change a
 - (c) if $e \neq 1$ and $e \neq n$ return d (it is a non-trivial factor of n)

There are 3 steps in this algorithm that would become too time-consuming to be able to use: computing $m = \text{lcm}(b)$, $a^m \pmod{n}$, and $d = \gcd(a^m - 1, n)$.

The first, compute $m = \text{lcm}(b)$, could be solved by making a list of all primes $\leq B$ by going through all integers smaller than B and checking for primality. However, this takes an extremely long time and could be replaced by the Sieve of Eratosthenes. To use the sieve, we write all the integers between 2 and the bound B . Then, for each number up to \sqrt{B} which is still remaining, we cross out all its multiples, at most an n number of p terms. The numbers remaining will be the integers smaller than B which have no factor smaller than \sqrt{B} , that is the primes smaller than B . The total operation required is:

$$\sum_{p=2}^{\sqrt{B}} \left\lceil \frac{B}{p} \right\rceil \approx \int_2^{\sqrt{B}} \frac{B}{p} dp = B \cdot \ln(\sqrt{B}) = O(B \cdot \log(B))$$

The second time-consuming step, calculating $a^m \pmod{n}$, can be calculated with fast modular multiplication. We compute it by expanding m as a sum of powers of 2, repeatedly squaring a , then multiplying the relevant powers of a .

Let $m = k_0 2^0 + k_1 2^1 + \dots + k_r 2^r$, where the k_i 's are either 1 or 0. Then, $a^m = a^{k_0 2^0 + k_1 2^1 + \dots + k_r 2^r}$ is calculated to $a^m = A_0 \cdot A_1 \cdot \dots \cdot A_r$. The A_i 's are computed by $A_0 = a$, $A_1 = A_0^2 = a^2$, \dots , $A_r = A_{r-1}^2 = a^{2^r}$. We only need r operations to compute the A_i 's and then at most r operations to add them together and get a^m . And, since $m = k_0 2^0 + k_1 2^1 + \dots + k_r 2^r \geq 2^r$, we get that $r \leq \log_2 k$. So, with this method, we can compute a^m in at most $2 \log_2 k$ operations, where each operation consists of one multiplication and one reduction \pmod{n} .

The third step, computing $d = \gcd(a^m - 1, n)$, can be computed using Euclid's algorithm.

4.3 Number Field Sieve

The number field sieve is an algorithm used to factor large numbers, usually meaning numbers with over 110 digits. It is one of the most efficient classical methods for factoring, which means it works on conventional computers. The number field sieve breaks down the problem of factoring into smaller, more manageable steps, allowing it to factor large numbers more quickly than other methods. There is also a "special" number field sieve, which is faster than the generalized version when factoring integers in the form of $r^e \pm s$ with $r, e, s \in \mathbb{Z}$ and $e > 0$.

The algorithm involves several crucial steps, including sieving, linear algebra, and polynomial selection. In the sieving step, a set of numbers is generated and filtered to find values that have a special property related to the factorization process. The linear algebra phase solves a system of equations using a matrix composed of the sieved numbers. Finally, polynomial selection is performed to find polynomials that satisfy certain conditions and aid in factoring.

This method can be read about more in the paper *A Beginner's Guide To The General Number Field Sieve* [Cas], but will not be covered in this expository paper due to length and extensive background needed.

The best implementation of this algorithm currently has a running time of $((c + o(1))n^{1/3} \log^{2/3} n)$.

4.4 Quantum Computing & Shor's Algorithm

Quantum computing poses a unique challenge, being able to factor large numbers very easily and quickly. The primary concern arises from Shor's algorithm, a quantum algorithm capable of efficiently factoring large numbers by utilizing the principles of quantum superposition and entanglement. If a practical quantum computer capable of running Shor's algorithm efficiently becomes available, it could undermine the security provided by RSA and other similar encryption methods.

Shor's algorithm is the current best algorithm for factorization. However, it cannot be used on prime numbers, even numbers, or numbers of the form x^z .

To use Shor's algorithm:

1. Choose a number, N , to factor
2. Randomly choose a number, k , between 1 and N
3. Calculate $\gcd(N, k)$
 - (a) If \gcd is not equal to 1, you have found a factor
 - (b) If \gcd is equal to 1, move on to step 4
4. We need to find the smallest possible integer r such that if $f(x) = k^x \pmod{N}$, then $f(a) = f(a + r)$
 - (a) Define a new variable $q = 1$
 - (b) Calculate $(q \cdot k) \pmod{N}$
 - i. If the remainder is 1, proceed to step 4.c
 - ii. If the remainder is not 1, set the value of q to the remainder. Repeat step 4.b until the remainder is 1.
 - (c) The number of transformations done is the value of r
5. If r is odd, go back to step 2 and choose a different value for k
6. Define p as the remainder in the $(r/2)$ th transformation
 - (a) If $p + 1 = N$, go back to step 2 and choose a different value for k
 - (b) If not, proceed to step 7
7. The factors of N are $f_1 = \gcd(p + 1, N)$ and $f_2 = \gcd(p - 1, N)$

Steps 1, 2, 3, 5, 6, and 7 can be run efficiently on a classical computer. However, step 4 takes a much larger amount of time, unrealistic for any current classical computer. In classical computers, this algorithm takes an exponentially larger amount of time with larger numbers. However, in quantum computers, the relationship is logarithmic. This makes Shor's algorithm extremely powerful, and potentially dangerous for large RSA keys.

Currently, there are a few claims for quantum related factoring.

- Xu, et al. [Xu+12] claimed to factor $143 = 11 \cdot 13$.
- Dattani, et al. [DB14] claimed to have factored $56153 = 233 \cdot 241$.
- Dash, et al. [Das+18] expanded on the previous claim and factored $4088459 = 2017 \cdot 2027$. This is similar to the previous claim. The method only applies to a very narrow class of integers (product of integers differing only by 2 bits). Both attempts fit the method. It also expands on Xu, et al. as their technique factors an integer product of two odd exactly-4-bit integers (thus with two unknown bits per factor). The scarcity of primes in range [8,15] makes 143 the only product of two distinct primes that the technique can factor. Their experimental setup iteratively minimizes a function with a 2-bit input.
- Pal, et al. [Pal+19] claims to have factored $551 = 19 \cdot 29$, but acknowledges that the technique probably can't factor some larger 10-bit integers. The number of qubit, which influences what integers can be factored, depends on the molecule used, in this case dibromofluoromethane.

- Li, et al. [Li+17] used a different molecule, 3C -labeled diethyl-fluoromalonate to factor $291311 = 523 \cdot 557$. They still use 3 qubits, but the larger value is explained by pre-screening the integer factored.

Any of the above factorizations can be obtained using the first step of Fermat's method:

1. let n be the integer to factor
2. let $u = \lceil \sqrt{n} \rceil$
3. if $v = \sqrt{u^2 - n}$ is an integer, then $u - v$ and $u + v$ are factors of n

None of the above techniques implement Shor's algorithm. They express factorization as a combinatorial minimization problem, solved using a variant of Grover's algorithm or adiabatic quantum computing. The approach is not scalable to anything that could hope to break a real RSA code, and is therefore of no cryptographic interest.

However, a recently released approach by Yan, et al. [Yan+22] is hopeful. By performing the harder part of the algorithm (Short Vector Problem) on a 10-bit quantum computer, they are able to factor any 48-bit integer. This raises the current record to $261980999226229 = 15538213 \cdot 16860433$.

References

- [SIM83] GUSTAVUS J. SIMMONS. “A “WEAK” PRIVACY PROTOCOL USING THE RSA CRYPTO ALGORITHM”. In: *Cryptologia* 7.2 (1983), pp. 180–182. DOI: [10.1080/0161-118391857900](https://doi.org/10.1080/0161-118391857900). eprint: <https://doi.org/10.1080/0161-118391857900>. URL: <https://doi.org/10.1080/0161-118391857900>.
- [DEL84] JOHN M. DELAURENTIS. “A FURTHER WEAKNESS IN THE COMMON MODULUS PROTOCOL FOR THE RSA CRYPTOALGORITHM”. In: *Cryptologia* 8.3 (1984), pp. 253–259. DOI: [10.1080/0161-118491859060](https://doi.org/10.1080/0161-118491859060). eprint: <https://doi.org/10.1080/0161-118491859060>. URL: <https://doi.org/10.1080/0161-118491859060>.
- [JQ87] M. Joye and J.J. Quisquater. “Faulty RSA encryption”. In: (Jan. 1987).
- [Hås88] Johan Håstad. “Solving Simultaneous Modular Equations of Low Degree”. In: *SIAM J. Comput.* 17 (1988), pp. 336–341.
- [Cop97] Don Coppersmith. “Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities”. In: *J. Cryptology* 10 (1997), pp. 233–260. DOI: [10.1007/s001459900030](https://doi.org/10.1007/s001459900030).
- [How97] Nicholas Howgrave-Graham. “Finding small roots of univariate modular equations revisited”. In: *Cryptography and Coding*. Ed. by Michael Darnell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 131–142. ISBN: 978-3-540-69668-1.
- [Ble98] Daniel Bleichenbacher. “Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1”. In: *Advances in Cryptology — CRYPTO ’98*. Ed. by Hugo Krawczyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–12. ISBN: 978-3-540-68462-6.
- [BDF98] Dan Boneh, Glenn Durfee, and Yair Frankel. “An Attack on RSA Given a Small Fraction of the Private Key Bits”. In: *Advances in Cryptology - ASIACRYPT ’98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings*. Vol. 1514. Lecture Notes in Computer Science. Springer, 1998, pp. 25–34. DOI: [10.1007/3-540-49649-1_3](https://doi.org/10.1007/3-540-49649-1_3).
- [Cha05] Anne-Sophie Charest. “Pollard’s p-1 and Lenstra’s factoring algorithms”. In: (Oct. 2005). URL: <https://www.math.mcgill.ca/darmon/courses/05-06/usra/charest.pdf>.
- [Xu+12] Nanyang Xu et al. “Erratum: Quantum Factorization of 143 on a Dipolar-Coupling Nuclear Magnetic Resonance System [Phys. Rev. Lett. b108/b, 130501 (2012)]”. In: *Physical Review Letters* 109.26 (Dec. 2012). DOI: [10.1103/physrevlett.109.269902](https://doi.org/10.1103/physrevlett.109.269902). URL: <https://doi.org/10.1103/2Fphysrevlett.109.269902>.
- [DB14] Nimesh S. Dattani and Nathaniel Bryans. *Quantum factorization of 56153 with only 4 qubits*. 2014. arXiv: [1411.6758](https://arxiv.org/abs/1411.6758) [quant-ph].
- [Li+17] Zhaokai Li et al. *High-fidelity adiabatic quantum computation using the intrinsic Hamiltonian of a spin system: Application to the experimental factorization of 291311*. 2017. arXiv: [1706.08061](https://arxiv.org/abs/1706.08061) [quant-ph].
- [Das+18] Avinash Dash et al. *Exact search algorithm to factorize large biprimes and a triprime on IBM quantum computer*. 2018. arXiv: [1805.10478](https://arxiv.org/abs/1805.10478) [quant-ph].
- [Pal+19] Soham Pal et al. “Hybrid scheme for factorisation: Factoring 551 using a 3-qubit NMR quantum adiabatic processor”. In: *Pramana* 92.2 (Jan. 2019). DOI: [10.1007/s12043-018-1684-0](https://doi.org/10.1007/s12043-018-1684-0). URL: <https://doi.org/10.1007/2Fs12043-018-1684-0>.
- [Bar20] Elaine Barker. “Recommendation for Key Management: Part 1 - General”. In: *NIST Special Publication* (May 2020). DOI: [10.6028/NIST.SP.800-57pt1r5](https://doi.org/10.6028/NIST.SP.800-57pt1r5). URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf>.
- [Yan+22] Bao Yan et al. “Factoring integers with sublinear resources on a superconducting quantum processor”. In: *arXiv preprint arXiv:2212.12372* (2022).
- [Cas] Michael Case. “A Beginner’s Guide To The General NumberField Sieve”. In: (). URL: <https://www.cs.umd.edu/~gasarch/TOPICS/factoring/NFSmadeeasy.pdf>.