The RSA cryptosystem

Zehra Kolat

MATH 411 Capstone Project 2

Advisor: Doğa Can Sertbaş

Istinye University
Faculty of Engineering and Natural Sciences
Mathematics Department

January 2025

The RSA cryptosystem

Zehra Kolat

27/05/2025

Abstract

In this project, we examine two important integer factorization algorithms in the context of breaking RSA encryption: Pollard's $p-1$ method and Wiener's attack. Pollard's $p-1$ algorithm is a number theoretic integer factorization algorithm. , while Wiener's attack targets RSA implementations with a small special exponent. The Wiener's attack, named after cryptologist Michael J. Wiener, the attack uses continued fraction representation to expose the private key d when d is small.

Acknowledgement

I would like to thank my advisor, Dr. Doğa Can Sertbaş, for the guidance and support he provided throughout this study. His expertise and helpful feedbacks were crucial for my academic progress on this research. I'm deeply grateful for his provided opportunities, guidances and countless helps for me academically and professionally.

Contents

1 Introduction

Modern cryptosystems rely on the difficulty of factoring large numbers. RSA, one of the most widely used public-key cryptosystems, derives its security from the difficulty of factoring a large composite number. However, certain factorization methods can exploit weaknesses in key generation or parameter selection to break the security. This project focuses on two such methods: Pollard's $p-1$ algorithm and Wiener's attack. Our goal is to survey these attacks and describe the underlying mathematical tools they use. Throughout the survey we follow standard naming conventions. Pollard's $p-1$ method is effective when B-smooth for a small bound B. Wiener's attack uses instances of RSA where the secret key d is very small. Using continued fraction approximations, it can efficiently compute d under certain conditions.

2 All codes covering capstone 1 topics

2.1 Key-Gen

We know about key-gen, encryption and decryption in RSA. The following code is for key-gen:

```
    import random
from sage.all import gcd, inverse_mod, power_mod, random_prime

# Function that selects prime numbers
def generate_large_primes(n):

    lower_bound = 2**((n-1)//2)
    upper_bound = 2**(n//2)

    # choose prime between 2^{(n-1)/2} and 2^{n/2}
    p = random_prime(upper_bound -1, False, lower_bound) #The value false indicates that an integer prime number will be selected.
    q = random_prime(upper_bound -1, False, lower_bound)

    while p == q:  # If p and q are the same, the loop continues until a new q is chosen.
        q = random_prime(upper_bound -1, False, lower_bound)

    return p, q


def choose_e(L):
    while True:
        e = random.randint(2, L - 2)  # Choose a random value between 2 and L -2
        if gcd(e, L) == 1:  # Choose e as long as it is gcd(e, L) = 1
            return e

# Function to generate RSA keys
def generate_RSA_keys(n):

    p, q = generate_large_primes(n)

    N = p * q
    L = (p - 1) * (q - 1)

    e = choose_e(L)

    d = inverse_mod(e, L)

    return (N, e), (N, d)  # public key ve private key


# Example for n = 1024 generate RSA keys
n = 1024
public_key, private_key = generate_RSA_keys(n)

print(f"Public Key: {public_key}")
print(f"Private Key: {private_key}")
```

Here is the output produced for $n = 16$ in the code:
$PublicKey : (52891, 21543)$
$PrivateKey : (52891, 49655)$

2.2 Encryption & Decryption

The following code is for Encryption & Decryption:

```

def encryption(x, pk):
    N, e = pk

    y = power_mod(x, e, N)

    return y

message = 14 # a sample message to be encrypted

encrypted_message = encryption(message, public_key)
print(f"Encrypted Message: {encrypted_message}")

def decryption(y, sk):
    N, d = sk

    z = power_mod(y, d, N)

decrypted_message = decryption(encrypted_message, private_key)
print(f"Decrypted Message: {decrypted_message}")
```

Here is the output produced for $n = 16$ & $message = 14$ in the code:
$PublicKey : (187, 79)$
$PrivateKey : (187, 79)$
$EncryptedMessage : 147$
$DecryptedMessage : 14$

2.3 Pollard Rho Algorithm

The following code is for Pollard Rho Algorithm:

```
    from random import randint
from sage.all import gcd

def pollards_rho(N):
    if N % 2 == 0:
        return 2

    def f(x, N):
        return (x**2 + 1) % N

    x = randint(1, N - 1)  # Z_N'den random x sectim.
    y = x
    i = 0

    while True:
        i += 1
        x = f(x, N)
        y = f(f(y, N), N)

        print(f"Iteration {i}: x = {x}, y = {y}")

        g = gcd(abs(x - y), N)

        if g > 1 and g < N:
            return g
```

```
        elif g == 1:
            continue
        else:
            return "failure"

# Example
N = 1517
factor = pollards_rho(N)
print(f"divisor: {factor}")
```

Here is the output produced for $N = 1517$:

$Iteration1 : x = 1249, y = 526$
$Iteration2 : x = 526, y = 82$
$divisor : 37$
An another example for $N = 1555989746988458748965 69$:
$Iteration1 : x = 796969979349977337907 53, y = 135056534629720426162920$
$Iteration2 : x = 135056534629720426162920, y = 142839707611084207059139$
$Iteration3 : x = 337580163447073287694 90, y = 756137793465025588223 18$
$Iteration4 : x = 142839707611084207059139, y = 954052492005420994787 43$
$divisor : 19$

2.4 Dixon's Algorithm

The following code is for Dixon's Algorithm:

```
    from sage.all import *
import random

def optimal_B_without_u(N):

    lnN = log(N).n()

    B_estimate = exp(sqrt((Integer(1)/Integer(2)) * (lnN * log(lnN).n())).n())

    B = next_prime(round(B_estimate))
    print(f"B estimate: {B_estimate}")
    print(f"B estimate round: {B}")
    return B

def dixon_factorization_without_u(N):
    B = optimal_B_without_u(N)  # B'yi otomatik belirle
    factor_base = list(primes(B))

    for p in factor_base:
        if N % p == 0:
            return p

    A = []  # (b, factorization) set storing pairs

    while len(A) < len(factor_base) + 1:
        b = random.randint(1, N-1)
        g = gcd(b, N)

        if g > 1:
            return g

```

```
        a = power_mod(b, 2, N)

        factorization = []
        for p in factor_base:
            exponent = 0
            while a % p == 0:
                a //= p
                exponent += 1
            factorization.append(exponent % 2)

        if a == 1:
            A.append((b, factorization))

    matrix_A = Matrix(GF(2), [alpha for _, alpha in A])
    null_space = matrix_A.right_kernel().basis()

    for solution in null_space:
        x = 1
        y = 1

        for i, coeff in enumerate(solution):
            if coeff == 1:
                x *= A[i][0]
                for j, prime in enumerate(factor_base):
                    if A[i][1][j] == 1:
                        y *= prime

        x = mod(x, N)
        y = mod(sqrt(y), N)

        g = gcd(x + y, N)
        if 1 < g < N:
            return g

        g = gcd(x - y, N)
        if 1 < g < N:
            return g

    return "failure"

# Test
N = 1545879895645
factor = dixon_factorization_without_u(N)
print("Found factor:", factor)
```

Here is the output produced for $N = 1545879895645$:

$$Bestimate : 935.144858889478$$
$$Bestimateround : 937$$
$$Foundfactor : 5$$

3 Pollard $p-1$ Algorithm

The factors found by the algorithm are the factors for which $p-1$ is powersmooth (where $N = pq$) ; the key observation is while composite N operates on a multiplicative group modulo N, it also operates on all factors of N modulo multiplicative groups. This makes it easier to find factors with certain properties (such as powersmoothness). The algorithm works when $p-1$ is B-smooth for any prime factor p of N. By Fermat's Little theorem, we know that for all integers a coprime to p and for all positive integers k:

$$a^{(p-1)k} \equiv 1 \pmod p$$

If a number x is congruent to 1 modulo a factor of n, then the $gcd(x-1, n)$ will be divisible by that factor. Thus $gcd(a^{(p-1)}-1, N) = p$. In summary, what we are basically saying is this:

$$a^{(p-1)k} \equiv 1 \pmod p \Rightarrow gcd(a^{(p-1)k} - 1, N) = p$$

The idea here is to make the number $p-1$ a product of small prime numbers (i.e. B-smooth).

Pollard's $p-1$ Factoring Algorithm
Input: A composite number N
Output: A nontrivial factor of N, or failure

1. Select a smoothness bound B.
2. define
$$M = \prod_{\text{primes } q \leq B} q^{\lfloor \log_q B \rfloor}$$
3. Set $a = 2$ (a small integer coprime to N).
4. Compute: $g = \gcd(a^M - 1, N)$
5. If $1 < g < N$, return g as a nontrivial factor.
6. If $g = 1$, increase B and repeat from step 2.
7. If $g = N$, decrease B, then repeat form step 2.

If g =1 in step-6, implies the number $p-1$ is not B-smooth or the prime factors of $p-1$ do not overlap the factors covered by the powers of the prime factors of M.

If $g = n$ in step-7, this usually indicates that all factors were B-powersmooth, but in rare cases it could indicate that a had a small order modulo N.

A rare case where the algorithm may fail: If for each of the prime factors of N, the largest prime factor of $p-1$ is the same, then the algorithm may fail.

3.1 Methods of choosing B

We know the smallest prime factor $p \leq \sqrt{N}$ from the number theory. So, we have:

$$p - 1 < p \leq \sqrt{N} \Rightarrow p - 1 < \sqrt{N}$$

By the *Dickman function* [7], if the largest factor of $p-1$ $(= B) \leq (p-1)^{1/\epsilon}$, this probability is $P \approx \epsilon^{-\epsilon}$.

$B \leq (p-1)^{1/\epsilon}$ implies $\Rightarrow 1/\epsilon = \log_{p-1} B$

$\Rightarrow 1/\epsilon = \log B / \log(p-1)$

$\Rightarrow \epsilon = \log(p-1)/\log B$ And we have, $p - 1 \leq \sqrt{N}$ if we combine all,

$(p-1)^{1/\epsilon} \leq N^{1/(2\epsilon)} \Rightarrow B \leq N^{1/(2\epsilon)}$

$\Rightarrow B = N^{1/(2\epsilon)}$

If we choose $\epsilon = 3$, then $B = N^{1/6}$ with 3.7% success rate. Also we choose $\epsilon = 2$ with 25% success rate, but it would be so costly. Why is this election so important? Because the success of the $p-1$ algorithm depends on the probability that $p-1$ is B-smooth.

3.2 Why Do We Calculate M?

We define $M = \prod_{(\text{primes } q \leq B)} q^{\lfloor \log_q B \rfloor}$ but why?

Our goal is to compute $a^M \pmod{N}$ and we know $a^{p-1} \equiv 1 \pmod{p}$. We want to use this to find p. If $p-1$ is a B-smooth number, $a^{p-1} \pmod{p}$ is easy to calculate. So we need to compute an exponent M that includes the divisors of $p-1$. M is desired to be a multiple of $p-1$ i.e. $M \equiv 0 \pmod{p-1}$. If this is achieved then $a^M \equiv 1 \pmod{p}$ and from here, we find the proper factor ($= p$) with $gcd(a^M - 1, N)$.

To determine M, we use powers of prime numbers less than B. We need to calculate the largest power q of $p-1$. $M = \prod_{(\text{primes } q \leq B)} q^{\lfloor \log_q B \rfloor}$ means for all prime numbers less than B, this means takes them to the largest possible power. For example, Let's calculate M when $B = 10$. The primes $q \leq 10$ are: $2, 3, 5, 7$.

- For $q = 2$, $\lfloor \log_2 10 \rfloor = 3$, so $2^3 = 8$.
- For $q = 3$, $\lfloor \log_3 10 \rfloor = 2$, so $3^2 = 9$.
- For $q = 5$, $\lfloor \log_5 10 \rfloor = 1$, so $5^1 = 5$.
- For $q = 7$, $\lfloor \log_7 10 \rfloor = 1$, so $7^1 = 7$.

Thus, the product M is:

$$M = 2^3 \cdot 3^2 \cdot 5^1 \cdot 7^1 = 8 \cdot 9 \cdot 5 \cdot 7 = 2520$$

Therefore, for $B = 10$, $M = 2520$.

3.2.1 Why does M contain all the factors of $p-1$?

For the algorithm to be successful, we assume that $p-1$ is the B-smooth number. And we know that $q^{\lfloor log_q B \rfloor} \leq q^{log_q B} = B$
that is, M is the product of prime numbers less than B. Thus, M is also a B-smooth number. Then if both $p-1$ and M are B-smooth numbers, then M contains all the factors of $p-1$.

So we are now sure that $a^M \equiv 1 \pmod p$ and thus $gcd(a^M - 1, N) = p$

! **Note that:** If any factor of $p-1$ is $> B$, then M does not contain those prime factors. Therefore $a^M \not\equiv 1 \pmod p$ and $gcd(a^M - 1, N) = 1$

So the goal of M is actually to capture the factors of $p-1$, and this is only possible if $p-1$ is B-smooth.

3.3 Time Complexity

The time complexity of this algorithm is $O(B \cdot logB \cdot log^2 N)$; larger values of B make it run slower, but are more likely to produce a factor.

The algorithm computes $a^M \pmod N$, that is, performs modular exponentiation, which takes approximately $O(log^2 N)$ time. The algorithm finding the largest powers of all prime numbers up to B, which means $\prod q^{\lfloor \log_q B \rfloor}$ it takes approximately $O(B \cdot logB)$ time.

3.4 The Code for Pollard $p-1$ algorithm

Here is a sagemath code of Pollard's $p-1$ algorithm.

```python
import random
from math import log, floor

def pollard_pminus1(N):

    if N % 2 == 0:
        print("Please select an odd composite number")
        return None

    B = next_prime(int(N ** (1 / 6))) # smoothness bound
    print(f"B: {B}")

    M = 1 # Initial value
    for q in primes(B): # For primes q up to B
        M *= q**floor(log(B, q))
    print(f"M: {M}")

    # Select a number a
    a = random.randint(1, N - 1)
    while gcd(a, N) != 1:
        a = random.randint(1, N - 1)

    print(f"a: {a}")
    g = gcd(power_mod(a, M, N) - 1, N)

    # Check steps 5, 6, and 7
    if 1 < g < N:
        return g  # A non-trivial factor found
    elif g == 1: # g = 1, increase B and repeat
        print("g == 1, select a larger B.")
        return None
    elif g == N: # g = N, decrease B and repeat
        print("g == n, select a smaller B.")
        return None

# Test
N = 1517 #  Enter a value of N that you want to try

# Try to find a factor
factor = pollard_pminus1(N)
if factor:
    print(f"Found prime factor: {factor}")
else:
    print("Failed.")
```

Listing 1: Pollard's p-1 Factorization Algorithm

If we run the code and try it a few times for $N = 1517$, we might see the following in the output:

Run 1:

B: 5
M: 12
a: 29
Found prime factor : 37

or **Run 2:**

B: 5
M: 12
a: 547
Found prime factor : 37

or **Run 3:**

B: 5
M: 12
a: 79
g == 1, select a larger B.
Failed.

4 The Wiener Attack

It is an attack that makes it possible to find d by using its relation to e when a very small d is chosen in the algorithm. As we know according to the RSA rules, the encryption exponent e is chosen at ran dom in $\{2, \cdots, \phi(N) - 2\}$ and the decryption exponent d so that $de = 1$ in $Z_{\phi(N)}$. Since modular exponentiation takes time linear in $log_2 d$, a small d can improve performance by at least a factor of 10 (for a 1024 bit modulus). Unfortunately, a clever attack due to M. Wiener shows that a small d results in a total break of the cryptosystem [9]. RSA computations are cheaper with small exponents, but one must be careful of the possible dangers.

We have $de - k \cdot \phi(N) = 1$ (where k is unknown positive integer).

If we divide by $d \cdot \phi(N) \Rightarrow \frac{e}{\phi(N)} - \frac{k}{d} = \frac{1}{d \cdot \phi(N)}$
$\phi(N)$ has approximately the same size of N, so we can write:

$$\frac{e}{N} - \frac{k}{d} \approx \frac{1}{d \cdot N}$$

If d is small, the ratio k/d gives e/N with an error rate of approximately $1/dN$ Since this error ratio is clearly smaller than $1/d$ $(1/dN < 1/d)$, the ratio e/N can be approximated with a small error.so if d is small enough, we have an approximation of unknown k/d to the known quantity N/e, with an error of $1/dN$. In this case, continued fractions are used to obtain a correct convergence of the value of k/d. This can be processed with the Euclidean algorithm to find the value of d.

4.1 Extended Euclidean Algorithm

Theorem 1 (Extended Euclidean Algorithm)**.** *(a, b) is a linear combinition of a and b: for some integers s and t,*

$$(a, b) = sa + tb$$

Proof. The Euclidean algorithm for integers a and b with $a > b > 0$ proceeds via the sequence:

$$\begin{aligned} a &= q_0 b + r_0 \\ b &= q_1 r_0 + r_1 \\ r_0 &= q_2 r_1 + r_2 \\ &\vdots \\ r_{N-2} &= q_N r_{N-1} + 0 \end{aligned}$$

This can be written as a product of 2×2 quotient matrices multiplying a two-dimensional remainder vector

:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_0 \end{pmatrix} \begin{pmatrix} b \\ r_0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \cdots = \begin{pmatrix} a \\ b \end{pmatrix} = \left(\prod_{i=0}^{N} \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \right) \begin{pmatrix} r_{N-1} \\ 0 \end{pmatrix}$$

Let M represent the total matrix product be:

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \prod_{i=0}^{N} \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & -q_N \end{pmatrix}$$

Then:

$$\begin{pmatrix} a \\ b \end{pmatrix} = M \begin{pmatrix} r_{N-1} \\ 0 \end{pmatrix} = \mathbf{M} \begin{pmatrix} g \\ 0 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} g \\ 0 \end{pmatrix} = \mathbf{M}^{-1} \begin{pmatrix} a \\ b \end{pmatrix}$$

Using the inverse of $\mathbf{M}$:

$$\mathbf{M}^{-1} \begin{pmatrix} a \\ b \end{pmatrix} = (-1)^{N+1} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

Thus:

$$\begin{pmatrix} g \\ 0 \end{pmatrix} = (-1)^{N+1} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

This gives Bézout's identity:

$$g = (-1)^{N+1}(m_{22}a - m_{12}b)$$

So the coefficients s and t such that $g = sa + tb$ are:

$$s = (-1)^{N+1}m_{22}, \quad t = (-1)^N m_{12}$$

□

[10]

4.2 Continued Fractions

A continued fraction is a mathematical expression that can be written as a fraction with a denominator that is a sum that contains another simple or continued fraction. Depending on whether this iteration terminates with a simple fraction or not, the continued fraction is finite or infinite [11]. If the sequence of convergents approaches a limit, the continued fraction is considered convergent and has a definite value.

For example, the number $\frac{415}{93}$ is represented as continued fraction as follows,

$$\frac{415}{93} = 4 + \cfrac{1}{2 + \cfrac{1}{6 + \cfrac{1}{7}}}$$

Theorem 2. *Any finite simple continued fraction represents a rational number. Conversaly, any rational number can be expressed as a finite simple continued fraction. [2]*

Proof. $\Leftarrow$:

Let r be a rational number like $r = \frac{p}{q}$ where $p, q \in \mathbb{Z}$. Firstly,

$$p = a_0q + r_1 \quad \text{for} \quad 0 \leq r_1 \leq q, a_0 = \lfloor \frac{p}{q} \rfloor$$

(by the euclid algorithm)

$$\frac{p}{q} = a_0 + \frac{r_1}{q}$$

now we have 2 case:

case1: If $r_1 = 0 \Rightarrow \frac{p}{q} = a_0$ thus $r = \frac{p}{q}$ is an integer and continued fractions ended with a_0.

case2: If $r_1 \neq 0 \Rightarrow \frac{r_1}{q} = \frac{1}{q/r_1} \Rightarrow \frac{p}{q} = a_0 + \frac{1}{\frac{q}{r_1}}$

Now we will do the same for $\frac{q}{r_1}$: (since $r_1 \neq 0$, q/r_1 is rational)

$$q = a_1r_1 + r_2 \quad \text{for} \quad 0 \leq r_2 \leq r_1, a_1 = \lfloor \frac{q}{r_1} \rfloor$$

So,

$$\frac{q}{r_1} = a_1 + \frac{r_2}{r_1}$$

Similarly, we have 2 cases as $r_2 = 0$ and $\neq 0$. If $r_2 = 0$, continued fraction ends in the same way. If not,

$$\frac{p}{q} = a_0 + \frac{1}{a_1 + \frac{r_2}{r_1}}$$

We repeat the same steps over and over again. This process ends in finite steps, because at each step the remainder decreases between positive integers such that $r_1 > r_2 > r_3 > \cdots \geq 0$. That is, at some point the remainder reaches 0. Thus, if r is rational then continued fraction is finite.

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}}$$

$\Rightarrow$:

Every finite simple continued fraction produces a rational number: We will show that any finite simple continued fraction corresponds to a rational number.

Let the continued fraction be

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

where $a_0 \in \mathbb{Z}$ and $a_i \in \mathbb{Z}^+$ for $i \geq 1$.

We evaluate the expression from the innermost term, recursively.

Define:

$$F_n = a_n$$

$$F_{n-1} = a_{n-1} + \frac{1}{F_n}$$

$$F_{n-2} = a_{n-2} + \frac{1}{F_{n-1}}$$

$$\vdots$$

$$F_0 = a_0 + \frac{1}{F_1}$$

Each F_k is formed by a finite number of additions and divisions involving integers, and since the set of rational numbers $\mathbb{Q}$ is closed under addition and division (except by zero, which doesn't occur here because $a_i > 0$), each F_k is a rational number.

Therefore, $F_0 = x$ is a rational number

$\square$

Lemma 3. *In the Extended Euclidean Algorithm applied to (a, b), the remainder at step i can be expressed as*

$$r_i = s_i a + t_i b.$$

Moreover, the coefficients satisfy the identity:

$$s_i t_{i+1} - t_i s_{i+1} = (-1)^i.$$

Proof. We have (i), (ii)

$$R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}, R_i \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$$

This follows directly from the definition of the matrix R_i, so the first clause of this lemma is verified using clauses (i) and (ii).

We know also the update matrix in the Extended Euclidean Algorithm: $Q_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix}$ and the determinant of this matrix is: $\det(Q_i) = (0)(-q_i) - (1)(1) = -1$. Therefore, we conclude that:

$$s_i t_{i+1} - t_i s_{i+1} = \det \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix} = \det R_i = \det Q_i \cdots \det Q_1 \cdot \det R_0$$

$$= \det Q_i \cdots \det Q_1 \cdot \det \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix} = \det Q_i \cdots \det Q_1 \cdot \det \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = (-1)^i$$

This statement shows that the (s_i, t_i) and (s_{i+1}, t_{i+1}) vectors form a linearly independent and "unimodular" matrix. A unimodular matrix is a square matrix (same number of rows and columns) with integer entries and determinant equal to ± 1. Unimodular matrices are invertible, and their inverse is also an integer matrix. Thus $gcd(s_i, t_i)$ and $gcd(s_{i+1}, t_{i+1})$ is ± 1, i.e. they are independent($\det \neq 0$). □

Lemma 4. *Let a and b be integers with $a > b \geq 0$, and suppose that $r, s, t \in \mathbb{Z}$ satisfy*

$$r = sa + tb \geq 0 \quad \textit{and} \quad 4r|t| \leq a.$$

Furthermore, define $i \leq l+1$ by $r_i \leq 2r < r_{i-1}$. Then there exists an integer u with $1 \leq |u| < r_{i-1}/2r_i$, $r = u \cdot r_i$, $s = u \cdot s_i$, and $t = u \cdot t_i$.. If $gcd(s, t) = 1$, then $u \in \pm 1$.

Proof. It is known from Lemma 3 that

$$r_i = s_i a + t_i b,$$

and also

$$r = sa + tb.$$

Then

$$r_i t - r t_i = (s_i t - s t_i)a + b(t_i t - t t_i) = (s_i t - s t_i)a.$$

This expression shows that a divides $r_i t - r t_i$.

Now let's look at the inequalities:

$$|r t_i| < r \cdot \frac{a}{2r} = \frac{a}{2}$$

cause $r_i \leq 2r < r_{i-1}$ and $|r_i t| \leq 2r|t| \leq a/2$ was given as. ($4r|t| \leq a$ due to the situation).

Thus

$$|r_i t - r t_i| \leq |r_i t| + |r t_i| < \frac{a}{2} + \frac{a}{2} = a.$$

a divides this difference $a \mid (r_i t - r t_i)$, and the difference is less than a in absolute value $|r_i t - r t_i| < a$, which is only possible if the difference is 0,

$$r_i t - r t_i = 0.$$

Then,

$$s_i t = s t_i.$$

Since it is known from Lemma 3 that $gcd(s_i, t_i) = 1$, the number t_i divides t and the number s_i divides s.

Thus there exists an integer u such that:

$$t = u t_i, \quad s = u s_i.$$

And also,

$$r = sa + tb = u(s_i a + t_i b) = u r_i.$$

And then,

$$|u| = \frac{|r|}{r_i} < \frac{r_{i-1}}{2r_i}.$$

If $\gcd(s, t) = 1$ then u is a common divisor

$$u = \pm 1.$$

as desired. □

Theorem 5. *Let $a > b$ and $s, t \in \mathbf{Z}_{>0}$ be positive integers such that $gcd(s,t) = 1$ and*

$$|\frac{b}{a} - \frac{s}{t}| \leq \frac{1}{4t^2}$$

Then either $(s, -t)$ or $(-s, t)$ appears as some (s_i, t_i) in the Extended Euclidean Algorithm applied to (a, b).

Proof. Let $t^* = -t$ and so $r = |sa + t^*b| = |sa - tb|$. If $r = 0$, in this case, $sa = tb$, that is, $s/t = b/a$, which is perfect equality.$c = gcd(a, b)$, and by Lemma 15.21 (iv), (s,t) is one of the coefficients occurring in the EEA. The theorem is provided.

And let ϵ is the sign of s_{l+1} and $gcd(a, b) = c$, then $gcd(\frac{a}{c}, \frac{b}{c}) = 1$, and form Lemma 3, we know $s_i a + t_i b = r_i$ (this also holds for $i - 1 = l$), so we find $s = \frac{b}{c} = \epsilon s_{l+1}$ and $t = \frac{a}{c} = -\epsilon t_{l+1}$ which shows (s, t) is a signed version of (s_{l+1}, t_{l+1}), proving the claim.

We may now assume $r > 0$, so $r = |sa - tb| > 0$. Then

$$4r|t^*| = 4|sa - tb| \cdot |t| = 4\frac{|sa - tb|}{at} \cdot at \cdot t = 4at^2|\frac{b}{a} - \frac{s}{t}| \leq 4at^2 \cdot \frac{1}{4t^2} = a.$$

This is one of the conditions of Lemma 4. This Lemma says that if $r = sa + t^*b \geq 0$ and $4r|t^*| \leq a$, then $(s, t^*) \in \pm(s_i, t_i)$ for some i, from which the claim follows.

□

Theorem 6 (Wiener's attack)**.** *Suppose $p < q < 2p$, $1 \leq e < \phi(N)$ and $1 \leq d \leq N^{1/4}/\sqrt{12}$. Then d can be computed from the public data in time $O(n^2)$.*

Proof. Using $\frac{e}{N} - \frac{k}{d} \approx \frac{1}{d \cdot N}$, we define r as:

$$r = KN - de = KN - (1 + k \cdot \phi(N) = KN - 1 - k \cdot \phi(N)$$

$$k \cdot \phi(N) - de - k \cdot (\phi(N) - N) = k \cdot \phi(N) - 1 - k \cdot \phi(N) - k \cdot \phi(N) + kN = -1 - k\phi(N) + kN$$

$$= -1 - k \cdot (N - (p + q) + 1) + kN$$

(because we know $\phi(N) = (p - 1)(q - 1) = N - (p + q) + 1$)

$$= -1 + k \cdot (p + q - 1)$$

As a result I now have:

$$r = KN - de = -1 + k \cdot (p + q - 1)$$

where $k \leq d$ and $k > 0$

Note: If $k > d$, then $1 = de - \phi(N)k < 0$, so it is not logically valid.

$r > 0$ from $k \leq d$ and $k > 0$ and $r = -1 + k(p + q - 1) > 0 \Rightarrow k(p + q) > k(p + q - 1) > 1$

$\Rightarrow 0 < r < k(p + q)$

We know $p < q < 2p \Rightarrow p + q < p + 2p = 3p$:

$0 < r < k(p + q) < 3kp$ and we know $k \leq d$ so, $0 < r < k(p + q) < 3kp < 3dp$ and we also know $p \leq N^{1/2} : 0 < r < k(p + q) < 3kp < 3dN^{1/2}$

This inequality says that for small values of d, k and therefore r will be small.We want to use theorem 1 to obtain a larger upper bound using the magnitudes of r and d.

$r < 3dN^{1/2}$ and also $4dr < 12rdN^{1/2} \Rightarrow 4dr < 12rdN^{1/2} \leq N$

can write less than N because $d \leq N^{1/4}/\sqrt{12} \Rightarrow d^2 \leq N^{1/2}/12 \Rightarrow 12d^2 \leq N^{1/2}$

And we know $N > e$, now we have everything we need to use theorem1:

$$|\frac{e}{N} - \frac{k}{d}| = |\frac{ed - kN}{Nd}| = \frac{1}{Nd}|ed - kN| = \frac{1}{Nd} \cdot r$$

now we can write $\frac{r}{Nd} \leq \frac{N}{4d \cdot Nd}$ from $4dr \leq N$

$$\Rightarrow |\frac{e}{N} - \frac{k}{d}| = \frac{r}{Nd} \leq \frac{1}{4d^2}$$

If this inequality holds, the difference between k/d and e/N will be very small, allowing us to find the correct value of d with the extended euclidean algorithm. When these inequalities are satisfied, using the Bezout's identity it is guaranteed that $gcd(k, d) = 1$ between k and d. So now we can say $gcd(s, t) = gcd(k, d) = 1$ using theorem 5. Thus, extended euclidean algorithm helps us find correct k, d (i.e.s, t) pairs when working on e and N. Now, $s \cdot e + t \cdot \phi(N) = 1$ in here s,t are Bezout's identity. □

4.3 Low Private Exponent

Using a small value of d in RSA speeds up signing and decryption because modular exponentiation works on $log_2 d$. However, this leads to a security vulnerability. $|\frac{e}{N} - \frac{k}{d}| \leq \frac{1}{4d^2}$ in Theorem 5 is a classic approximation relation. The number of fractions $\frac{k}{d}$ with $d < N$ approximating $\frac{e}{N}$ so closely is bounded by $log_2 N$. In fact, all such fractions are obtained as convergents of the continued fraction expansion of $\frac{e}{N}$. All one has to do is compute the $logN$ convergents of the continued fraction for $\frac{e}{N}$. One of these will equal $\frac{k}{d}$. Since $ed - k\phi(N) = 1$, we have $gcd(k, d) = 1$, and hence $\frac{k}{d}$ is a reduced fraction. This is a linear-time algorithm for recovering the secret key d.

The attacker can efficiently calculate d by knowing only N and e. Basic Idea of Wiener's Theorem: Since $ed \equiv 1 \mod \phi(N)$, there exists a k such that:

$ed - k\phi(N) = 1 \Rightarrow |\frac{e}{\phi(N)} - \frac{k}{d}| \approx \frac{1}{d \cdot \phi(N)}$. In this case, $\frac{k}{d}$ is a fraction very close to $\frac{e}{\phi(N)}$. However, $\phi(N)$ is unknown, so an approximation of N can be used instead. This convergence relation allows finding k using continued fractions. The continued fraction expansion of $\frac{e}{N}$ is calculated. Approximately $logN$ convergents are examined. One will give k/d and in this case d is captured.

4.3.1 What Should Be Done For Security?

Since typically N is 1024 bits, it follows that d must be least 256 bits long in order to avoid this attack (if N is 1024 bits, then $N \approx 2^{1024}$ and we know $d \leq N^{1/4}/\sqrt{12} \Rightarrow d \leq 2^{1024/4}/\sqrt{12} = 2^{256}/\sqrt{12}$, since $\sqrt{12}$ is so small, it can be ignored). This is a problem for small devices (smart cards, IoT devices) because they require small d to work fast. But for security reasons, a minimum of 256 bits should be set. All is not lost however. Wiener offers a number of alternatives that provide fast decryption and are invulnerable to attacks:

Large e: In RSA, Wiener's attack can recover the private key d efficiently when $d < \frac{1}{\sqrt{12}}N^{1/4}$, using continued fractions and the approximation $\frac{k}{d} \approx \frac{e}{\phi(N)}$.

To prevent this, one can choose a large public exponent:

$$e' = e + t \cdot \phi(N), \quad \text{where } t \text{ is a large integer.}$$

Normally, the e part of the public key is chosen to be smaller than $\phi(N)$. But here we choose a very large e' specifically. Then the public key becomes (N, e'), and encryption still works correctly.

The wiener attack uses the approximation $k/d \approx \frac{e}{\phi(N)}$. However, if e is very large, then: the ratio $k/d \approx \frac{e'}{\phi(N)}$ is also very large. so k is also very large.

Why it works: If $e' > N^{1.5}$, then the approximant $\frac{k}{d}$ is no longer close enough to $\frac{e'}{\phi(N)}$, so Wiener's attack fails—even if d is small. However, large values of e make the encryption time longer.

Trade-off: Larger e' increases encryption time, since modular exponentiation takes time proportional to $\log_2 e'$.

The Wiener attack is no longer possible because mathematical closeness is broken. That is, normally small d: risky (because Wiener works); but very large e': no longer risky (because Wiener doesn't work).

We do not know if any of these methods are secure. All we know is that Wiener's attack is ineffective against them. The theorem 4 was developed recently by Boneh and Durfee.

4.4 Calculation of d

Therefore $d = u \cdot t_i$,where $u = \pm 1$ and t_i is one of the entries in the Extended Euclidean algorithm with inputs N and e. We take some $x \in \mathbb{Z}_{\mathbb{N}}^*$ (means $gcd(x, N) = 1$) and $x \neq \pm 1$. In practice we can choose $x = 2$ because N is a composite odd number and gcd is strictly 1 without requiring checking. Now check this:

if $x^{et_i} = x = x^{-(et_i)}$ is provided then $x^{2et_i} = x^{\pm 2} = 1$ is also provided. Since $x \neq \pm 1$, this actually gives us the factorization of N, hence d. Otherwise, exactly one of $x^{et_i} = x$ or $x = x^{-(et_i)}$ holds, and this gives us the value of d.

4.5 The Code for Wiener's Attack

Here is a sagemath code of Wiener's Attack algorithm:

```

e = 53387
N = 82123

test_values = [2]

ratio = e / N

cf = continued_fraction(ratio)

convergents = cf.convergents()

print("convergents:", convergents)

for conv in convergents:
    k = conv.numerator()
    d = conv.denominator()
    print(f"Convergent: k = {k}, d = {d}")


    ed = e * d
    valid = True

    for x in test_values:
        x_pow = power_mod(2, ed, N)
        x_neg_pow = power_mod(2, -ed, N)
        x_mod = x % N

        if x_pow == x_mod or x_neg_pow == x_mod:
            x_double_exp = power_mod(2, 2 * ed, N)
```

```
            if x_double_exp == (x_pow * x_pow) % N or x_double_exp == (x_neg_pow * x_neg_pow) % N:
                print(f"x = {x}: x^(ed) = x veya x^(-ed) = x The squared conditions x mod N and x^(2ed) mod N are satisfied.")
            else:
                print(f"x = {x}: Square condition not met -> x^(2ed) mod N = {x_double_exp}")
                valid = False
                break
        else:
            print(f"x = {x}: Conditions not met -> x^(ed) = {x_pow}, x^(-ed) = {x_neg_pow}, x mod N = {x_mod}")
            valid = False
            break

    if valid:
        print(f"\n Found d valid : d = {d}")
        break
else:
    print("\n No valid d found.")
```

when we run this code the output is :
Valid d found: d = 3

References

[1] Ivan Niven & Herbert S. Zuckerman & Hugh L. Montgomery, An Introduction to the theory of numbers, 5th Edition, page 325 *Simple Continued Fractions*

[2] Ivan Niven & Herbert S. Zuckerman & Hugh L. Montgomery, An Introduction to the theory of numbers, 5th Edition, page 329 *Theorem 7.2*

[3] Wiener's Attack/ Joachim von zur Gathen, CryptoSchool.

[4] G.H.Hardy & E.M. Wrıght, An Introduction to the theory of numbers, 4th Edition, page 129 *Finite Continued Fractions*

[5] Neal Kobiltz, Number Theory and Cryptography, second edition.

[6] David B. Burton, Elementary Number Theory, 7th Edition.

[7] Wikipedia. Available at: `https://en.wikipedia.org/wiki/Dickman_function`

[8] Pollard $p-1$ Algorithm/ Wikipedia, available at: `https://en.wikipedia.org/wiki/Pollard%27s_p_%E2%88%92_1_algorithm`

[9] Twenty Years of Attacks on the RSA Cryptosystem / Dan Boneh. Available at: `https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf`

[10] Wikipedia/Matrix Method of Extended Euclidean Algorithm Available at: `https://en.wikipedia.org/wiki/Euclidean_algorithm`

[11] Wikipedia. Available at: `https://en.wikipedia.org/wiki/Continued_fraction`

[12] Cryptanalysis of RSA with Private Key d Less Than $N^{0.292}$ Available at: `https://staff.emu.edu.tr/alexanderchefranov/Documents/CMSE491/Fall2019/BonehIEEETIT2000%20Cryptanalysis%20of%20RSA.pdf`