Pollard's Rho Algorithm: An In-Depth Analysis and Practical Applications in Large Integer Factorization, Discrete Logarithmic Problems, and SHA-256 Collisions

Ai Zhengjie
School of Computer Science and Engineering
Nanyang Technological University, Singapore
zai001@e.ntu.edu.sg

Song Changlin
School of Computer Science and Engineering
Nanyang Technological University, Singapore
s220177@e.ntu.edu.sg

Tao Zifeng
School of Computer Science and Engineering
Nanyang Technological University, Singapore
taoz0007@e.ntu.edu.sg

Tian Yunxuan
School of Computer Science and Engineering
Nanyang Technological University, Singapore
tian0140@e.ntu.edu.sg

***Abstract*—Pollard's rho method is a Monte Carlo algorithm for finding nontrivial factors of large integers, proposed by mathematician John Pollard in 1975. Its main purpose is to help find factors of large numbers in fields such as cryptography and digital security, thus solving the problem of inefficiency of traditional factorization methods. Compared with the traditional trial division method, Pollard's rho method utilizes the ideas of randomness and iterative computation to significantly improve the efficiency of factorization. In this report, we will introduce the steps and algorithmic details of Pollard's rho method, as well as demonstrate its importance and advantages through three practical applications: large integer factorization, discrete logarithmic problems, and collision of SHA-256.**

***Index Terms*—Pollard's Rho, Factorization of large integers, Discrete logarithmic problems, Collision of SHA-256**

CONTENTS

I. PERSONEL

TABLE I
GROUP INFORMATION

Name	Student ID
Ai Zhengjie	G2304717E
Song Changlin	G2204828F
Tao Zifeng	G2304600D
Tian Yunxuan	G2304372L

II. Introduction

In modern cryptography, safeguarding message security is of paramount importance. The RSA(Rivest-Shamir-Adleman) algorithm, renowned for its asymmetric encryption capabilities, plays a pivotal role in ensuring the confidentiality of communications, upholding data integrity, and establishing secure connections. Its security relies on the formidable challenge of factoring large integers or solving discrete logarithmic problems.

The Pollard's rho algorithm was introduced by the British mathematician John Pollard. This algorithm is inspired by the product decomposition theorem (also known as a generalization of Fermat's Little Theorem), which states that if p is a prime number, a is an integer, and a is not a multiple of p, then $a^{(p-1)} \equiv 1(\text{mod}p)$ Pollard's rho algorithm utilizes this property to find non-trivial factors of integers.

The basic idea of the algorithm is to generate sequences by choosing a starting point at random and using a specific iterative function. If there is a loop in the sequence, then a non-trivial factor in the loop can be found by computing the greatest common factor of the sequence.

The origin of the name "Pollard's Rho" is based on the similarity in appearance between the Greek letter ρ (rho) and the sequence that the algorithm creates, as shown in Figure 1. The tail length (t) is the number of steps before we enter a cycle, and the cycle length (s) is the length from the start of the cycle until we get a collision.

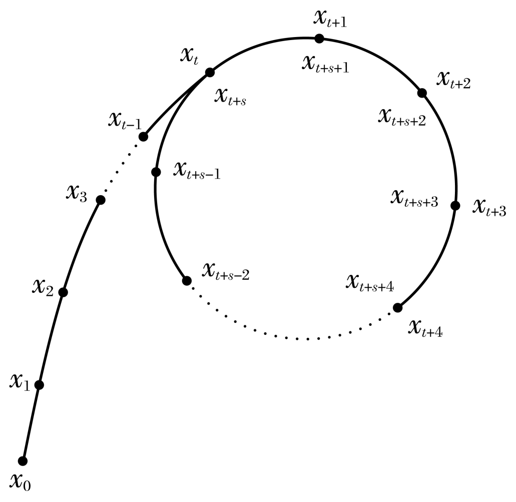

Fig. 1. Pollard's Rho

The input to the algorithm is the points P of the elliptic curve over the field F_q. These points are divided into L sets, called branches, of approximately the same size, by using a partition function f. If f is a random function, we can make some assumptions about how long time it will take before we start the cycle, i.e. the tail length t and the expected cycle length s. We have $t \approx \sqrt{\frac{\pi n}{2}}$ that terms and $s \approx \sqrt{\frac{\pi n}{8}}$ where n is the modulo.

Pollard's rho algorithm is widely used in practice to factorize large integers, and in particular it plays an important role in security analysis and attacks on RSA encryption algorithms. It is an efficient identification of non-trivial factors in large integers, thereby reducing computational complexity. Its existence also emphasizes the need to choose a sufficiently large prime number for security when designing cryptosystems, and has far-reaching implications for ensuring communication security and cryptographic research.

A. RSA Factoring Challenge

RSA is a now widely used public key cryptosystem proposed by Ron Rivest et al. in 1977.The RSA system is designed with separate public and private keys, where the public key is distributed to the encryptor for encrypting the message, and the private key, which is undistributed and needs to be kept hidden, is used for decrypting the message. In addition to encryption, RSA can also be used for digital signatures.

The RSA can be divided into four steps: key generation, key distribution, message encryption, and message decryption.

Select two prime numbers p, q. Calculate

$$N = pq$$

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

Pick an integer e that is mutually prime with $\phi(N)$. According to

$$ed \equiv 1 \ (mod \ \phi(N))$$

d can be calculated by the Extended Euclidean algorithm.

The public key consists of (n, e) and the private key consists of (n, d), the public key will be transmitted to the encryptor, who encrypts the message with the public key and transmits it to the decryptor. And the private key d will not be distributed and the decryptor uses the private key to decrypt the ciphertext.

For the message m to be encrypted, compute

$$c = (m^e) \mod n$$

c is the ciphertext after encryption of the plaintext m.

Decrypt the message using (n, d) in the private key,

$$m = (c^d) \mod n$$

m is the decrypted result of the ciphertext c.

In the above steps, anyone holding the public key can encrypt the message and deliver the ciphertext, but only the person holding the private key can decrypt the ciphertext to get the plaintext, this design ensures the security of the system.

RSA factorisation refers to the decomposition of the prime factors p , q in $N = pq$ from a known N. The difficulty of this procedure is that it is easy to multiply two large prime factors p , q to obtain the ensemble product N, however, factoring the ensemble N to produce p , q without any other hints is difficult, and it is generally recognised that there is no known efficient algorithm that can solve this problem in polynomial time. The difficulty of factoring large numbers ensures the security of RSA being difficult to crack.

In order to encourage research into integer factorisation and cryptography, the RSA Factoring Challenging has published a series of RSA semiprimes to be broken for researchers to try to break. The smallest of these, the hundredth decimal number RSA-100, was successfully cracked in 1991. The latest result is in 2020, Fabrice Boudot et al [2] decomposed RSA-250 (829 bits), which took about 2700 CPU core-years, while the decomposition of 1024-bit RSA is expected to take 100 times longer.

TABLE II
PARTIAL HISTORY OF RSA FACTORING

RSA number	Decimal digits	Binary digits	Factored on
RSA-100	100	330	April 1, 1991
RSA-120	120	397	July 9, 1993
RSA-155	155	512	August 22, 1999
RSA-576	174	576	December 3, 2003
RSA-640	193	640	November 2, 2005
RSA-768	232	768	December 12, 2009
RSA-240	240	795	Dec 2, 2019
RSA-250	250	829	Feb 28, 2020

B. Discrete Logarithm

Discrete logarithms play a crucial role in cryptography, particularly in public key cryptographic systems and key exchange protocols. The discrete logarithm problem refers to the difficulty of finding a discrete logarithm within a given finite group.

The description of the discrete logarithm problem is as follows[1]:

Suppose there is a group

$$(G, \cdot)$$

, where

$$\alpha \in G$$

is an element of order n. Given

$$\beta \in \langle \alpha \rangle$$

, the task is to find an exponent

$$c$$

, with

$$0 \leq c \leq n - 1$$

, such that

The security of public key cryptographic systems relies on the complexity of the discrete logarithm problem. Common encryption algorithms such as the Diffie-Hellman key exchange protocol and ElGamal signature scheme are based on this problem. These systems leverage the complexity of the discrete logarithm problem to ensure the security of communication since solving the discrete logarithm problem requires a significant amount of computation, which current computing technology cannot reasonably solve within a feasible timeframe in the domain of large integers.

C. Collision of SHA-256

SHA-256 is a cryptographic hash function, one of the SHA algorithms, and the successor of SHA-1. It was developed by the US National Security Agency and is an algorithm subdivided under SHA-2. For messages of any length, SHA256 will produce a 256-bit (32-byte array) hash value, called a message digest. The digest is usually represented as a 64-bit hexadecimal string.

SHA-256 is widely used in cryptography, blockchain, digital certificates, file integrity verification, etc.

Hash collision refers to the situation where two different input data obtain the same hash value after being calculated by the Hash function. Within the theoretical scope, there is an output string corresponding to infinite input strings, so collision is inevitable. If a collision is found, it means that we can destroy the consistency of the information without being noticed by the receiver. The process of searching for the Hash collision value of the specified input is called ”Hash cracking”. During the query and addition process of HashMap, if the calculated Hash values are the same, the two objects with the same calculation results need to be stored in the same linked list. This is the Hash collision in HashMap.

Here are the current collisions for all hash functions:

TABLE III
COMPARISON OF SHA FUNCTIONS

Algorithm and variant		Output size (bits)	Block size (bits)	**Security against collision attacks(bits)**
MD5 (as reference)		128	512	<=18 (collisions found)
SHA-0		160	512	<=34 (collisions found)
SHA-1				<=63 (collisions found)
SHA-2	SHA-224	224	512	112
	SHA-256	256		128
	SHA-384	384	1024	192
	SHA-512	512		256
	SHA-512/224	224		112
	SHA-512/256	256		128
SHA-3	SHA3-224	224	1152	112
	SHA3-256	256	1088	128
	SHA3-384	384	832	192
	SHA3-512	512	576	256
	SHAKE128	d(arbitrary)	1344	mid(d/2,128)
	SHAKE256	d(arbitrary)	1088	mid(d/2,256)

III. METHODS

A. Theory of Pollard’s rho

Consider a sequence such as $s_0 = a$,

$$g(x) = (x^2 + 1) \bmod n$$

where n is the number attempting to factorize. And the sequence continues as

$$x_1 = g(2), x_2 = g(g(2)), x_3 = g(g(g(2)))$$

Then generate in turn the triples where

$$(x_i, x_{2i}, Q_i), i = 1, 2, ...,$$

$$Q_i = |x_{2i} - x_i| \bmod n$$

Because the number of possible values for these sequences is finite, the sequence is ultimately periodic. As there are integers $c \geq 1$ and $t \geq 0$ such that $x_0, x_1, ..., x_{c+t-1}$ are all distinct $\mathrm{mod} p$, then finally we will find

$$x_{c+i} \equiv x_i (\mathrm{mod} p)$$

So we have

$$Q_i \equiv 0 (\mathrm{mod} p)$$

,which means we find a factor of n.

$$d_i = \gcd(Q_i, n)$$

If $1 < d_i < n$ then we have obtained a partial factorization of n as $n = d_i * (n/d_i)$.

Once a sequence has a repeated value, the sequence will cycle, because each value depends only on the one before it. This structure of eventual cycling gives rise to the name "rho algorithm", owing to similarity to the shape of the Greek letter ρ, are represented as nodes in a directed graph.

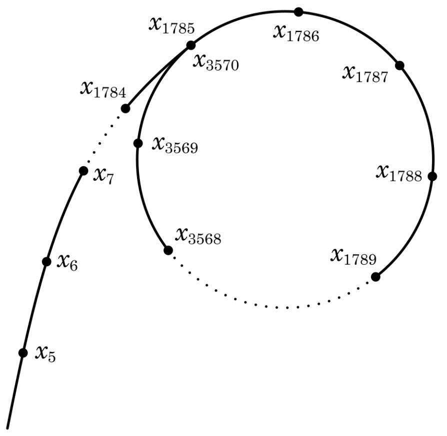

Fig. 2. Pollard's Rho

This is detected by Floyd's cycle-finding algorithm, here we are going to explain it.

Floyd's Cycle Detection Algorithm, also known as the "Tortoise and Hare Algorithm", is an algorithm for detecting the presence of a cycle in a linked table.

The core idea of Floyd's Cycle Detection Algorithm is to use two pointers, a slow pointer (the tortoise) and a fast pointer (the hare), which traverse a linked table or sequence from the same starting point. The fast pointer moves forward two steps at a time, while the slow pointer moves

$$x_i \bmod p = x_j \bmod p$$

forward one step at a time. If a loop exists, then the fast pointer will eventually catch up with the slow pointer, forming a loop.

The advantage of Floyd's Cycle Detection Algorithm is that it is a very simple and fast algorithm that can be executed in constant space without the need for additional data structures to hold the nodes that have been visited. This makes it a powerful tool for detecting the presence of loops in a linked table.

B. Pollard's Rho based RSA factorisation

On RSA Factoring, ordinary prime enumeration method is very time consuming, the best known algorithms are large integer based factoring techniques such as Pollard's rho algorithm, GNFS algorithm.

Pollard's rho based rsa factoring can reduce the number of greatest common divisor calculations, thus improving the computational efficiency, and can find the factor of n within $O(\sqrt{p})$ time complexity.

The idea is to know the N to be decomposed, set up two nodes x_i, x_j, and in each step, x_i moves one node at a time and x_j moves two nodes at a time, and then check whether $gcd(x_i - x_j, N) \neq 1$. If it is not equal to it, then the result of greatest common divisor will be p. And, it also implies that the sequence $x_k \mod p$ will be repeated as well. This is because, if $x_i \equiv x_j \mod p$, then the difference between x_i and x_j must be a multiple of p. This means that gcd will repeat if $x_i \equiv x_j (\mod p)$. Also, this shows that the result that $gcd(x_i - x_j, N)$ is not equal to 1 will come eventually anyway, but probably N itself, at which point it represents a failure of Pollard's ρ algorithm, which can be run again, using a different starting value.

Take $N = 10$ for example, randomly generate $x = 2$, choose $f(x) = x^2 + 1 \bmod N$, compute

step 1	step 2
$x = f(x) = 5$	$x = f(x) = 6$
$y = f(f(x)) = 6$	$y = f(f(x)) = 0$
$d = gcd(\|x - y\|, N) = 1$	$d = gcd(\|x - y\|, N) = 2$

due to $d \neq 1$ and $d \neq N$, it follows that d is a factor of N. We change N to N/d and continue start a new loop:

1) step 1
   - $x = f(x) = 0$
   - $y = f(f(x)) = 1$
   - $d = gcd(|x - y|, N) = 1$
2) step 2
   - $x = f(x) = 1$
   - $y = f(f(x)) = 0$
   - $d = gcd(|x - y|, N) = 1$
3) step 3
   - $x = f(x) = 2$
   - $y = f(f(x)) = 2$
   - $d = gcd(|x - y|, N) = 5$

At this point it can be shown that $(2, 5)$ is the solution (p, q) of N to be solved. This is a clear example, for a much larger one with a random 24-digit number N, which took 70s to complete on the tested device. The time required for general factorisation will grow with the number of digits of N.

C. Pollard's Rho algorithm for logarithm

Pollard's rho algorithm for logarithms is an algorithm introduced by John Pollard in 1978 for solving the discrete logarithm problem analogous to Pollard's rho algorithm for solving the Integer factorization problem. It is a heuristic algorithm, not a deterministic mathematical proof algorithm. It may yield good solutions, but it can also fail, depending on the initial parameters chosen by the attacker.

Algorithm

Let G be a cyclic group of order p, and given $\alpha, \beta \in G$
In G, and a partition $G = S_0 \cap S_1 \cap S_2$
Let $f : G \to G$ be a map

$$f(x) = \begin{cases} \beta x, x \in S_0 \\ x^2, x \in S_1 \\ \alpha x, x \in S_2 \end{cases}$$

and define maps $g : GXZ \to Z$and $h : GXZ \to Z$ by:

$$g(x,n) = \begin{cases} n, x \in S_0 \\ 2n(\text{mod} p), x \in S_1 \\ n+1(\text{mod} p), x \in S_2 \end{cases}$$

$$h(x,n) = \begin{cases} n+1(\text{mod} p), x \in S_0 \\ 2n(\text{mod} p), x \in S_1 \\ n, x \in S_2 \end{cases}$$

Here is a simple version of the Pollard's Rho algorithm for discrete logarithm presented in pseudocode

```
input: a: a generator of G
       b: an element of G
output: An integer x such that ax = b, or failure
     Initialise a0 = 0, b0 = 0, x0 = 1 belongs to G
i=1 loop
    xi = f(xi−1),
    ai = g(xi−1, ai−1),
    bi = h(xi−1, bi−1)
    x2i = f(f(x2i−2)),
    a2i = g(f(x2i−2), g(x2i−2, a2i−2)),
    b2i = h(f(x2i−2), h(x2i−2, b2i−2))
    if xi = x2i then
        r = bi − b2i
        if r = 0 return failure
        x = r−1(a2i − ai) mod n
        return x
else // xi != x2i i=i+1
    end if
end loop
```

The space requirements of algorithms using the rho method are negligible. Therefore to solve the DLP in groups of large group orders, this method is superior to Shanks' baby step-giant step method that has roughly the same runtime but space requirements $O = \sqrt{|G|}$.

D. Pollard's Rho for Hash collision

General steps for finding the first 80 identical SHA-256 hash collisions using Pollard's Rho algorithm:

1. Select initial value: Select two different initial input values. This will become the initial value for the "fast" and "slow" sequences. The initial value can be selected by generating a random string.

2. Iteration: The core idea of using Pollard's Rho algorithm is to generate the next input value through iteration. At each step, the "fast" and "slow" sequences are updated. Among them, the operation of each step of the "fast" sequence is to calculate the SHA-256 hash value twice (the result after each hash is only the first 80 bits), while the operation of the "slow" sequence is to calculate the SHA-256 hash value once (The hash result only takes the first 80 bits).

For example: h function represents SHA-256 hash calculation, and then only extracts the first 80 bits. Then the operation of each step of the "fast" sequence is $x = h(h(x))$, and the operation of each step of the "slow" sequence is $x = h(x)$.

3. Check for collisions: At each step, compare the calculated result values of the "fast" and "slow" sequences. If their values are the same, then a collision has been found.

4. After found collisions: Reset the "fast" sequence to the initial value and reset the step size of the "fast" sequence to one, that is, the operation of each step of the "fast" sequence is to perform a SHA-256 hash calculation, that is, $x = h(x)$, and then continue to iterate until the "fast" sequence collides with the "slow" sequence again. At this time, two different input values that calculated by SHA-256 have generated the same hash value of the first 80 bits.

5. Adjust iteration parameters: As needed, you can adjust iteration step length, initial value selection and other parameters to improve the collision search efficiency.

6. Repeat iterations: Pollard's Rho algorithm usually requires a large number of iterations to find a collision, so the above steps need to be repeated until a collision is found or the decision is made to give up.

The pseudocode for the above steps is:

```
function generate_random_string(length):
    characters =random string
    return certain length of characters

function sha256_80bit_hash(message):
    sha256 = create_sha256_object()
    sha256.update(message.encode('utf-8'))
    return first 80 bits of the hash value

function pollards_rho_collision():
    random_string = generate_random_string(64)
    x, y = random_string, random_string
    x_prev, y_prev = random_string, random_string
    while True:
        x_prev, y_prev, iteration = x, y, iteration + 1
        x, y = sha256_80bit_hash(x_prev),
            sha256_80bit_hash(sha256_80bit_hash(y_prev)
            )
        if x == y:
            y = generate_random_string(64)
            while True:
                y_prev, x_prev, x, y = y, x,
                    sha256_80bit_hash(x_prev),
                    sha256_80bit_hash(y_prev)
                if x == y:
                    return x, y, iteration , x_prev,
                        y_prev, sha256_80bit_hash(
                        x_prev), sha256_80bit_hash(
                        y_prev)

if __name__ == "__main__":
    cx, cy, iterations , xp, yp, sx, sy =
        pollards_rho_collision ()
```

The python code for the above algorithm can be found in appendix.

IV. Results

A. Pollard's Rho based RSA factorisation

Based on the Pollard's rho algorithm, a factorization test was conducted on increasing decimal N values. The results showed that the Pollard's rho algorithm can complete the RSA factorization task correctly and quickly, and the prime numbers p and q obtained were verified to be correct.

- 10 number
  - N:5693732377
  - p:85247
  - q:66791
- 15 number
  - N:292102426805677
  - p:34662259
  - q:8427103
- 20 number
  - N:23716309006544171183
  - p:5290105211
  - q:4483145053
- 25 number
  - N:1762288295707486437706103
  - p:2768781385613
  - q:636485171731
- 30 number
  - N:231352657657655271099126414823
  - p:323553015295859
  - q:715037866193597
- 35 number
  - N:16503967778886920110080805388357687
  - p:52020855346403077
  - q:317256755372209931
- 40 number
  - N:1342466574277580264380434227219748175457
  - p:33696939273368749117
  - q:39839421716813132021

Among these RSA numbers, RSA40 corresponds to a 117-bit binary number. It takes more than one day to complete the factoring task using a single thread on a personal computer. This is based on the premise of optimization by Pollard's rho algorithm, which shows the difficulty of RSA factoring task. In the discussion section, we have analysed the difference in time consumption before and after optimization by Pollard's rho algorithm to prove the advantage of this method in terms of time complexity.

B. Pollard's Rho algorithm for logarithm

The specific implementation of the experiment is carried out in the Python 3.9 environment, with practical improvements based on the algorithm mentioned in Section 2. Building upon the core of the Pollard rho algorithm, the key to solving the discrete logarithm problem lies in efficiently finding collisions, denoted as $\alpha^a\beta^b = \alpha^A\beta^B$ (referred to as collisions hereafter). The Pollard rho algorithm itself is a randomized approach. Therefore, during collision detection, randomization can be employed in the entry point (randomly selecting a, b). Simultaneously, choosing an appropriate detection depth minimizes the time cost to detect collisions effectively. This approach aims to achieve the lowest time cost in finding collisions and subsequently solving the discrete logarithm problem.

$$(a - A) * (B - b)^{-1}(\text{mod} p)\ 1)$$

Due to computational constraints, the large prime numbers chosen for this experiment range from 20 bits to 50 bits, denoted as p. Additionally, to find a suitable generator for constructing a cyclic group modulo the chosen prime, this experiment selects the smallest generator, denoted as g. Therefore, the problem addressed in the experiment can be described as follows:

$$find\ x : g^x = y \bmod p$$

Refer to the table below for specific experimental parameters.

Size of Prime(bits)	Prime p	g	y	result x
20	1015919	7	5	238670
30	749699189	2	5	351206112
40	104902656121	11	5	6708464224
50	159534088683653	2	5	40953714105525

For a more detailed analysis of the experiment, we selected Average iterations to collision ρ, Times of experiments, Times of successes, and Time consumption as analysis parameters. Here, ρ represents the average number of iterations required to find a collision (Note: the iteration count here only includes cases where collisions were successfully found and results were computed within a given number of loops, excluding cases where collisions were not found within the specified loop count). Since the algorithm uses randomization to accelerate collision searching, success rate is also recorded. In terms of time calculation, the experiment logs the total time under the corresponding large prime number. Refer to the table below for specific experimental data.

Size of Prime (bits)	Average iterations to collision ρ	Times of experiments	Times of successes	Time consumption(s)
20	1547	1000	734	9.84954
30	56378	1000	371	351.9179358
40	228424	1000	184	1734.1803779
50	18007278	100	14	10243.8168983

Within a given number of loops (detection depth), the number of collisions found, denoted as ρ, can be approximately considered as the square root of the cyclic group order,$\sqrt{p}$, which aligns with the predictions in the methodology section. As observed from the table above, with an increase in the bit length of large prime numbers, the probability of successfully finding collisions within a limited number of loops continuously decreases. For a 50-bit prime number, there is only a 14% probability of successfully finding a solution within a finite number of loops. Regarding computation time, as the bit length of the prime number increases by every 10 bits, the order of magnitude of the computation time also increases by one. Therefore, for encryption algorithms such as ElGamal that rely on the discrete logarithm problem, the security is considerably high at 2048 bits and 3072 bits.

C. Pollard's Rho for Hash collision

The experimental environment is as follows:

- PyCharm 2023.2.1 (Community Edition)
- Runtime version: 17.0.8+7-b1000.8 x86_64
- OpenJDK 64-Bit Server VM by JetBrains s.r.o.
- M1 chip with macos 14.1.1
- GC: G1 Young Generation, G1 Old Generation
- Memory: 1500M
- Cores: 8
- Metal Rendering is ON

In the experimental phase, a python program using Pollard's Rho algorithm developed for finding SHA-256 collisions was first used to try to find collisions in the first 20 bits of SHA-256. Finding collisions in the first 20 bits was very fast, and almost every time the program was run, different values could be found. (As the experiment progresses, we find that when the number of bits increases, the time to find collisions is very long, and the program running results are often the same, that is: the probability of finding the same collision increases), the following figure is a partial finding Display of the collision of the first 20 bits of SH-256:

TABLE IV
COLLISION FOR SHA-256 FIRST 20 BITS

Collision	Message 1	Message 2	Iterations used
263b7	fed6a	a337a	507
33011	84dc5	25bcf	261
8c437	a29ba	c7644	1014

Then, try to find the collision of the first 40 bits of SHA-256. When looking for the collision of the first 40 bits, the program still runs very quickly, and the collision can be found in an instant. However, we found that there are a lot of repetitions in the collision results after running the program multiple times, which requires It takes many repeated runs to find different collision results, which greatly increases the time required to conduct experiments to find collisions in the first 40 bits of SHA-256. The following are the results of the collisions found in the first 40 bits of SHA-256:

TABLE V
COLLISION FOR SHA-256 FIRST 40 BITS

Collision	Message 1	Message 2	Iterations used
c2f0fc6b3f	251ca3ba0d	cc3dd914b0	1897506
62ae47b6c6	9269994f32	623f3089e7	948753
f45aee77cc	a53df2d783	2b8309ce79	5494511

When the experiment progressed and tried to find the collision of the first 60 bits and the first 80 bits of SHA-256, the program ran for a long time without producing any results, so no experimental result data was obtained. The specific reason for the failure to obtain results has been clarified, because Pollard's Rho It is not an efficient algorithm for finding SHA-256 collisions, and in the process of using Pollard's Rho to find collisions, the message generated in each iteration lacks flexibility (for example, when looking for the collision of the first 40 bits of SHA-256, the message generated each time message is only 40 bits long), and finding collisions under a fixed bit length (the limitation of Pollard's Rho algorithm in finding SHA-256 collisions) will make the entire collision search process more difficult.

V. DISCUSSION

A. Comparison of Pollard's rho and other algorithms on RSA Factorization

In order to compare the advantages of Pollard's rho algorithm in terms of time complexity, we conducted an experiment and designed trial division algorithm and Fermat's factorization algorithm. Call these two algorithms to factorize the same RSAN, and compare the time required to calculate the result with Pollard's rho algorithm.

The results show that Pollard's rho algorithm with a time complexity of $O(\sqrt{n})$ is significantly better than the trial division algorithm and Fermat's factorization algorithm in terms of computational efficiency, showing the advantages of this algorithm in RSA Factoring.

The results show that the Pollard's rho algorithm with a time complexity of $O(\sqrt{n})$ is significantly better than the traditional method in terms of computational efficiency, showing the advantages of this algorithm in RSA Factoring.

- Binary digits:16
  - N(decimal):64529
  - p:373
  - q:173
  - Trial division:2 ms

  - Fermat's factorization:2 ms
  - Pollard's rho:2 ms
- Binary digits:32
  - N(decimal):3538564511
  - p:72167
  - q:49033
  - Trial division:17 ms
  - Fermat's factorization:2 ms
  - Pollard's rho:4 ms
- Binary digits:48
  - N(decimal):243164091795397
  - p:28994729
  - q:8386493
  - Trial division:2338 ms
  - Fermat's factorization:2883 ms
  - Pollard's rho:240 ms
- Binary digits:64
  - N(decimal):17462199092071723981
  - p:6526486381
  - q:2675589601
  - Trial division:1683473 ms
  - Fermat's factorization:442143 ms
  - Pollard's rho:644 ms

The difficulty of RSA factorisation shows its security, when the length of RSA's N is 512 bits or even larger, it is difficult to crack RSA on a PC even using the Pollard's rho algorithm. it is generally accepted that N of 2048 bits or more is secure. Now the common RSA public key certificate is 4096 bits.

In the future, with the development of quantum computing, it may become easy to crack RSA, and then the length of the RSA key may be increased or other quantum encryption methods may be used to improve the security of the public key encryption system.

B. Improvements on Pollard's rho algorithm

The Pollard Rho algorithm requires a long computation time when dealing with the discrete logarithm problem in cyclic groups formed by large prime numbers. Collision search can be used to optimize and improve its performance.

In collision search, the objective is to create an appropriate function f and find two different inputs that produce the same output. The selection of function f is aimed to serve some cryptanalytic purpose. To utilize the generalized rho method, it's required that f has the same domain and range ($e.g., f : S \to S$), and that f is sufficiently complex to exhibit behavior similar to a random mapping. A random mapping is one chosen uniformly across all possible mappings. To carry out a parallel collision search, each process or step proceeds as follows. Choose a starting point $x_0 \in S$ and generate a sequence of points $x_i = f(x_{i-1})$ for $i = 1, 2, ...$ until reaching a distinguished point x_d based on some easily testable distinguishing property, such as a fixed number of leading zero bits. Add this distinguished point to a common list for all processors and start generating a new sequence of points from a new starting point.

Based on the application of collision search, additional information needs to be stored with the distinguished point (for example, storing x_0 and d in order to quickly locate points a and b that satisfy $f(a) = f(b)$. When the same distinguished point appears twice in the central list, a collision is detected. Multiple processors traverse through set S in a pseudo-random manner, generating various trails that terminate at distinguished points. Once any trail intersects with another trail, they will coincide from that point onwards. Upon detecting the first collision, if more collisions are required, trails and distinguished points can be continuously generated until the desired number of collisions is found.

In this way, the whole process of Pollard rho algorithm for discrete logarithm will be speed up.

C. Discussion on Pollard's Rho for Hash collision

1) Optimization: Even if the collision conditions of sha256 are limited to the first 80 bits, it is extremely difficult to use Pollard's Rho algorithm to find collisions. The original algorithm needs to be optimized as much as possible:

Parallel computing: Since the memory footprint of this algorithm is extremely low, computing efficiency can be improved by using multiple threads for parallel computing at the same time.

Maximum number of iterations: Choose a maximum number of iterations to avoid falling into inefficient operations due to poor current randomly generated initial values.

2) Discussion of experimental results: In the experiment, as the number of bits trying to collide with SHA-256 increases, the time required to find collisions will become longer and longer, and it will become more and more difficult to find different collisions. In view of the characteristics of Pollard's Rho algorithm (when finding When entering the ring, the "fast" sequence needs to be run from the beginning and then enter the ring again to get a collision). When looking for collisions, the message used each time lacks randomness and flexibility to a large extent (the number of bits in the message is the same, and all Consists of letters and numbers and does not contain special characters), so using Pollard's Rho algorithm to find collisions is not the best solution. And when the number of bits increases, multiple experiments often produce the same results, which is not conducive to finding more collision instances.

VI. Conclusion

Through in-depth research on Pollard's Rho algorithm and using Pollard's Rho algorithm to solve three practical problems, we concluded that Pollard's Rho algorithm has the following characteristics:

- Randomness: Pollard's Rho algorithm is a randomized algorithm that relies on randomness to find collisions.
- Memory efficiency: Pollard's Rho algorithm has extremely low memory requirements. It retains only two states (x and y) at each step without the need to store the entire state space, which makes the algorithm extremely efficient when dealing with large-scale problems.

- Parallelism: Pollard's Rho algorithm is easy to parallelize. This algorithm performs well in distributed and parallel computing environments because each thread can generate and process different states independently and the algorithm uses very little memory per thread.
- Non-deterministic: Due to the stochastic nature of the algorithm, there is no guarantee that a collision will be found on every run. However, by increasing the number of iterations, the probability of finding a collision can be increased.
- Suitable for specific scenarios: Pollard's Rho algorithm is particularly suitable for discrete logarithm problems and factorization problems under certain specific circumstances. In these cases, it can provide high efficiency.
- Not suitable for all problems: Pollard's Rho algorithm is not suitable for hash collision problems, especially when the hash function has high collision resistance (such as the SHA-256 hash function we chose). In some cases, more complex algorithms may be needed to find collisions.

Overall, Pollard's Rho algorithm is a flexible and effective algorithm suitable for some specific problem areas, especially when randomness is required and when there are strict requirements for memory efficiency.

References

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

[2] Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., & Zimmermann, P. (2020). Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment. In Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40 (pp. 62-91). Springer International Publishing.

[3] McCurley, K. S. (1990). The discrete logarithm problem. In Proc. of Symp. in Applied Math (Vol. 42, pp. 49-74).

[4] Shanks, D. (1971). Class number, a theory of factorization, and genera. In Proc. Symp. Math. Soc., 1971 (Vol. 20, pp. 415-440).

[5] Van Oorschot, P. C., & Wiener, M. J. (1999). Parallel collision search with cryptanalytic applications. Journal of cryptology, 12, 1-28.

[6] Van Oorschot, P. C., & Wiener, M. J. (1994, November). Parallel collision search with application to hash functions and discrete logarithms. In Proceedings of the 2nd ACM Conference on Computer and Communications Security (pp. 210-218).

[7] Gilbert, H., & Handschuh, H. (2003, August). Security analysis of SHA-256 and sisters. In International workshop on selected areas in cryptography (pp. 175-193). Berlin, Heidelberg: Springer Berlin Heidelberg.

[8] Chabaud, F., & Joux, A. (1998, August). Differential collisions in SHA-0. In Annual International Cryptology Conference (pp. 56-71). Berlin, Heidelberg: Springer Berlin Heidelberg.

[9] Biham, E., & Chen, R. (2004). Near-collisions of SHA-0. In Advances in Cryptology–CRYPTO 2004: 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004. Proceedings 24 (pp. 290-305). Springer Berlin Heidelberg.

[10] Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (2018). Handbook of applied cryptography. CRC press.

[11] Pollard, J. M. (1978). Monte Carlo methods for index computation (mod p). Mathematics of computation, 32(143), 918-924.

Appendix

A. Python code for RSA factoring

```
import random

def gcd(a, b):  # Find the  least  common multiple
    while b:
        a, b = b, a % b
    return a

def pollard_rho(N):  # Apply pollard's rho algorithm
    def f(x):
        return (x**2 + 1) % N
    x = random.randint(1, N - 1)
    y = x
    d = 1
    while d == 1:
        x = f(x)
        y = f(f(y))
        d = gcd(abs(x - y), N)
    if d == N:
        return None
    return d

def factorize(N):  # Factoring N and output the factors
    factors = []
    while N > 1:
        factor = pollard_rho(N)
        if factor is None:
            factors.append(N)
            break
        while N % factor == 0:
            N //= factor
        factors.append(factor)
    return factors

N = 243164091795397 # input N of RSA
print("RSA N:", N)
factors = factorize(N)
print(factors)
```

B. Python code for discrete logarithm

```
import gmpy2
import time
import random


def ext_euclid(a, b):
    if b == 0:
        return a, 1, 0
    else:
        d, xx, yy = ext_euclid(b, a % b)
        x = yy
        y = xx - (a // b) * yy
        return d, x, y


def linear_congruence(a, b, m):
    gcd, x, y = ext_euclid(a, m)
    if b % gcd == 0:
        x0 = (x * (b // gcd)) % m
        return x0
    else:
        return None


def xab(x, a, b, G, H, P, Q):
    sub = x % 3  # Subsets

    if sub == 2:
        x = x * G % P
        a = (a + 1) % Q

    if sub == 1:
        x = x * H % P
        b = (b + 1) % Q

    if sub == 0:
        x = x * x % P
        a = a * 2 % Q
        b = b * 2 % Q

    return x, a, b


def pollard(g, y, p, output_file):
    # P: prime
    # H:
    # G: generator
    count = 0
    q = p - 1
    var1 = random.randint(1, 10000)
    var2 = random.randint(1, 10000)

    a, b = var1, var2
    x = (g ** a) * (y ** b)

    X = x
    A = a
    B = b

    with open(output_file, 'a') as file:

        for i in range(1, 50000000):
            x, a, b = xab(x, a, b, g, y, p, q)
            X, A, B = xab(X, A, B, g, y, p, q)
            X, A, B = xab(X, A, B, g, y, p, q)
            print(f"{i:3}  {x:4} {a:3} {b:3}  {X:4} {A
                :3} {B:3}")

            if x == X:
                print(f"{i:3}  {x:4} {a:3} {b:3}  {X:4}
                    {A:3} {B:3}")
                file.write(f"{i}\t|{x}\t|{a}\t|{b}\t|{X
                    }\t|{A}\t|{B}\n")
                if gmpy2.gcd(B - b, q) == 1:
                    solution = ((a - A) * pow(B - b,
                        -1, q)) % q
                else:
                    solution = linear_congruence(B - b,
                        a - A, q)
                print("condition:", g, "^", a, y, "^",
                    b, " = ", g, "^", A, y, "^", B)
                if solution is not None:
                    file.write(f"result is:{solution,
                        pow(g, solution, p) == y}\n")
                    if pow(g, solution, p) == y:
                        count = 1
                print(solution)
                return a, b, A, B, x == X, count, i

        return a, b, A, B, x == X


output_file = "result_50b_50000000_100_5.txt"
start_time = time.time()
count = 0
sum = 0
for i in range(100):
    # res = pollard(2, 781790663358367023559139,
        1152276167080204122055108, output_file)
    res = pollard(2, 5, 159534088683653, output_file)
    if res[4] and res[5]:
        count += 1
        sum += res[6]
        print("success")
    else:
        print("false")
print("Success number:", count)
print("Average iteration:", sum / count)
end_time = time.time()
total_time = end_time - start_time
print(f"Time: {total_time} seconds")
```

C. Python code for SHA-256 collision

```
import hashlib
import random
import string
import threading

def generate_random_string(length):
    characters = string.ascii_letters + string.digits
    random_string = ''.join(random.choice(characters)
        for _ in range(length))
    return random_string

def sha256_80bit_hash(message):
    sha256 = hashlib.sha256()
    sha256.update(message.encode('utf-8'))
    hex_digest = sha256.hexdigest()
    return  hex_digest [:15]   # Take the  first  80 bits  (20
        hex  characters )

def  pollards_rho_collision  () :

    random_string = generate_random_string (64)
    print (random_string)
    x = random_string
    y = random_string
    x_prev = random_string
    y_prev = random_string
    countmax=random.randint(1000000000,20000000000)
    iteration  = 0

    while True:
        x_prev = x
        y_prev = y

        iteration  += 1

        if  iteration >countmax:
            break
        x = sha256_80bit_hash(x_prev) #x_prev
        y = sha256_80bit_hash(sha256_80bit_hash(y_prev))
            #y_prev
        #print ( iteration )
        if x == y:
            print ("Break !!!!!!!!! ")
            y = random_string
            while True:
                y_prev = y
                x_prev = x
                x = sha256_80bit_hash(x_prev)
                y = sha256_80bit_hash(y_prev)
                if x == y:
            #if x != x_prev and y != y_prev :
            #if x_prev != sha256_80bit_hash(y_prev):
                    return x, y,  iteration ,x_prev,
                        y_prev,sha256_80bit_hash(x_prev
                        ),sha256_80bit_hash(y_prev)
    return  0,0,0,0,0,0,0

def mainrunpra():
    cx = 0
    #while cx==0:
        #print ("New begin  !!!!!!!!!!!!!!!  ")
    cx, cy,  iterations ,xp,yp,sx,sy =
        pollards_rho_collision  ()
    print (" Collision  found:  ")
    print ("sha256 80 bits :  ", cx)
    print (cy)
    print ("Message 1:  ",xp)
    print ("Message 2:  ",yp)
    print (sx)
    print (sy)
    print ( f" Iterations : { iterations }")
    with open(" result . txt ", "a") as  file :
        fmessage="---\n"+'\n'+ "Collision found: \n"+"
            sha256 80bits :  "+cx+'\n'+"Message 1: "+ xp+
            '\n'+"Message 2: "+ yp+'\n'+str( iterations )
            +'\n'+"--------------------"
        # file . write ( thread_id , " Collision  found:  ")
        file . write (fmessage)


if  __name__ == "__main__":
    mainrunpra()
    #threads = []
    #for i in range(1):
    #     thread = threading .Thread( target =mainrunpra,
        args=(i ,) )
    #     threads . append(thread )

    #for thread in threads :
    #     thread . start ()

    #for thread in threads :
    #     thread . join ()
    cx = 0
    # while cx==0:
    # print ("New begin  !!!!!!!!!!!!!!!  ")
    cx, cy,  iterations , xp, yp, sx, sy =
         pollards_rho_collision  ()
    print (" Collision  found:  ")
    print ("sha256 80 bits :  ", cx)
    print (cy)
    print ("Message 1:  ", xp)
    print ("Message 2:  ", yp)
    print (sx)
    print (sy)
    print (f" Iterations : { iterations }")

    print ("All threads have finished .")
```