Final Project Report: RSA and Factoring Algorithms
Mel Nguyen, Bohan Li

Abstract

The security of RSA depends on the difficulty of factoring numbers that are products of large primes. The difficulty of factoring is seemingly non-obvious. This report explores the process of attacking RSA through factoring. Here, RSA, along with two of the fastest factoring algorithms: Pollard's P-1 Algorithm and Quadratic Sieve, are implemented. Timing analysis is performed by generating random primes of sizes $m = 6, 11, 16, ..., 36$ bits long, calculating n as the product of two unique primes, and factoring n using both algorithms. Our results indicate that Pollard's algorithm is significantly faster than Quadratic Sieve but begins to fail for primes longer than 15 bits. Results also verify the fact that the runtime of Quadratic Sieve is exponential with respect to prime bit length despite being one of the fastest factoring algorithms. The slow speed of Quadratic Sieve and failure of Pollard's for larger primes is evidence to show that factoring is indeed difficult as n grows. Given that RSA utilizes 1024 bit to 4096 bit keys, it is easy to see that factoring these keys is computationally infeasible, thus rendering direct factoring of the keys useless as an attack on RSA.

Introduction

RSA (Rivest-Shamir-Adleman) is a cryptosystem developed in 1977 and has evolved to become one of the most popular cryptosystems due to its security and ease of use. The seemingly simple problem of factoring numbers is the key strength in why this system has remained unbroken. In this project we experiment with two of the fastest known factoring algorithms, quadratic sieve and Pollard's p-1 algorithm to see how they fare against primes of differing sizes. The project includes implementation of both factoring algorithms, an example of how RSA works, and a timing analysis of how long the algorithms take to factor products of primes of differing sizes. All of our code/implementation was written using Python 3.6 and should be run as such.

Quadratic Sieve Implementation

The quadratic sieve algorithm utilizes the principle that two numbers $x^2 \equiv y^2, x \neq y \ (mod\ n)$ yields a nontrivial factor given by $gcd(x-y, n)$. This algorithm optimizes the process for finding x and y. The algorithm takes a set of possible candidate numbers and runs them through a "sieve". The algorithm takes the following steps[1]:

[1] "Quadratic Sieve." *Wikipedia*, Wikimedia Foundation, 26 Mar. 2019, en.wikipedia.org/wiki/Quadratic_sieve.

1. Choose a smoothness bound B such that we search for numbers that are B-smooth. The goal is to find numbers that are strictly products of primes p less than B such that n is a quadratic residue mod p. Call the set of these primes the factor base. Define the exponent vector for one of these numbers a that is a product of primes in the factor base such that if the factorization of a is given by

$$a = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$$

Then the exponent vector of a is $(e_1, e_2, ..., e_k)$. In our implementation, $B = 10000$ was chosen.

2. Select some number of elements to sieve m, and find all B-smooth numbers. In our implementation, we start with $m = 100$.
3. For each smooth number, factor it and generate its exponent vector.
4. Use linear algebra to find a subset of these vectors that add to the zero vector mod 2. This subset of numbers represents numbers that are squares mod n and whose product is a square mod n, yielding a congruence $x^2 \equiv y^2 \ (mod\ n)$.
5. Compute $gcd(x - y, n)$ for each solution. If no nontrivial factor is found, double m and repeat steps 2-5.

Sieving was accomplished by selecting $0 \le x \le m$, and computing $y(x) = (x - h)^2 - n$ for $h = ceil(\sqrt{n})$. This is designed to make sure that y is a square mod n while minimizing the value of y, increasing the chance that y is B-smooth. The sieve is defined to be a vector V_s that contains all y-values, ordered by x-value. For each prime p in the factor base, compute

$$x \equiv \sqrt{n} - h \equiv a \ (mod\ p)$$

Since n is a quadratic residue mod p, $\sqrt{n} \ (mod\ p)$ is an integer and is computed via the Tonelli-Shanks algorithm[2]. For each a, $y(a + ip)$ is divisible by p for all $i \ge 0$. So, divide each value at index $a + ip$ by p. Once this process has been completed for all p, values that are 1 in the sieve indicate B-smooth values of y.

Once exponent vectors are generated, a matrix is generated containing all of the exponent vectors as rows, for which linear algebra is applied to find the solution[3].

---

[2] "Tonelli-Shanks Algorithm." *Tonelli-Shanks Algorithm - Rosetta Code*, rosettacode.org/wiki/Tonelli-Shanks_algorithm.

[3] Koc, Cetin K, and Sarath N. *A Fast Algorithm for Gaussian Elimination over GF(2) and Its Implementation on the GAPP*. Journal of Parallel and Distributed Computing, 1991.

Pollard's P-1 Implementation

Our implementation of Pollard's p-1 algorithm can be seen in the file 'pollards.py'. There are 2 methods in the file, one pollards(num) which takes in the number to factor and factor(num, b) which takes the previous number and a smoothness bound. Pollards(num) will run continuously until a factor is found, increasing or decreasing b depending on the value of gcd(a^m -1, n)

Some background about the algorithm, it is based on the fact that for a prime number 'p' and numbers coprime to p, 'a', then $a^{K(p-1)}$ is congruent to 1 mod p by Fermat's little theorem. Now suppose we have some number 'x' that is divisible by p-1, then x = (p-1)k. We then have this congruence:

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

Which indicates to us that:

$$a^{x} - 1 \equiv 0 \ (mod\ p)$$

Meaning p divides a^x - 1. Knowing this and the fact that p is a factor of N, gcd(a^x-1, N) includes p. Our algorithm follows this, choosing the coprime 'a' as 2 since our numbers will always be odd. There are a few ways of calculating 'x' which is called 'm' in our code. We choose m as the product of all primes less than the smoothness bound B. The higher B is, the more likely a factor will be found but also the longer the running time of the algorithm. The steps are as followed: [4]

1. Default smoothness bound b = 1
2. Generate list of all primes greater than 0, less than b.
3. For every prime in the list, q, calculate $q^{floor(\log_q b)}$ and multiply the results as m.
4. Default coprime a = 2, since any number to factor will be odd.
5. Computer g = gcd(a^m (mod N) - 1, N)
6. If g = 1, then none of the prime factors p where p-1 was b-powersmooth.
    a. Increase b and go to step 2
7. If g = n, all of the prime factors were b-powersmooth.
    a. Decrease b and go to step 2
8. If g was between 1 and n, non inclusive then that means that g is a factor of N and is returned.

There are optimizations that could be made to this algorithm, an important one is choosing the smoothness bound b, however we didn't think it was necessary since the

[4] "Pollard's P − 1 Algorithm." Wikipedia. April 12, 2019. Accessed May 02, 2019. https://en.wikipedia.org/wiki/Pollard's_p_−_1_algorithm.

point of this project is to show the difficulty of factoring primes. It should be noted however that the algorithm can constantly be running since b changes depending on the results. Meaning if a factor is not found, the program will continue and should be counted as a failure. Because of this our programs returns a failure if a factor is not found in 1000 iterations.

RSA Implementation

```
import utils
# This is an example of how RSA is implemented.

def genKeyPair(klen):
    e = 3 # Choose some e that is relatively prime to phi(n)
    p = utils.genPrime(klen//2, 40)
    q = utils.genPrime(klen//2, 40)

    while (p%e == 1):
        p = utils.genPrime(klen//2, 40)
    while (q%e == 1):
        q = utils.genPrime(klen//2, 40)

    N = p*q
    phi = (p-1)*(q-1)
    print("phi: {}".format(phi))
    d = utils.xgcd(e,phi)[1]
    if d < 0:
        d += phi

    return (N,e,d)

myRSA = genKeyPair(10)
print(myRSA)

N = myRSA[0]
e = myRSA[1]
d = myRSA[2]

message = 420

cipher = pow(message,e,N)

decrypt = pow(cipher,d,N)
print(decrypt)
```

```
Python 3.7.2 (tags/v3.7.2:9a3ff
(Intel)] on win32
Type "help", "copyright", "cred
>>>
========= RESTART: C:\Users\Ngu
phi: 616
(667, 3, 411)
420
>>>
========= RESTART: C:\Users\Ngu
phi: 616
(667, 3, 411)
420
>>>
```

Our implementation of RSA can be found in the file ‘rsa.py’. To run this file, simply open the .py file in an environment like IDLE and hit ‘F5’ or run -> run module. The implementation lies in the function genKeyPair which takes in an argument klen which is the desired bit length of the final key. The encryption key, ‘e’ is automatically set as 3, but it should be chosen as some value that is relatively prime to phi(n).

The primes, p and q, are determined by the function genPrime which takes arguments of prime bit length and rounds of primality testing. genPrime utilizes the Miller-Rabin test to generate a number and test for it's primality. This is probabilistic so the results will always be 'very likely prime'.

Once p, q are determined N and phi(N) are calculated. Using the extended Euclidean algorithm, the multiplicative inverse of e mod phi(N) is found which is the decryption key d. The function returns N, e, and d. After calling function genKeyPair, the script uses the given values to encrypt a message m = 420 and decrypts it, showing the result.

Comparison Results

For each bit length, 5 sets of random pairs of primes were chosen, and the maximum runtime was obtained for each bit length and algorithm.

Table 1: Maximum runtimes by bit length

Prime Bit Length	Max Pollard's Runtime (s)	Max QuadSieve Runtime (s)	# Pollard's Failures	# QuadSieve Failures
6	0.001	4.19767	0	0
11	0.02194	8.8501	0	0
16	1.09507	3.19386	2	0
21	0.0559	19.18537	3	0
26	0.07394	22.7205	4	0
31	0	46.31431	5	0
36	0	113.40801	5	0

Figure 1: Runtime of Quadratic Sieve

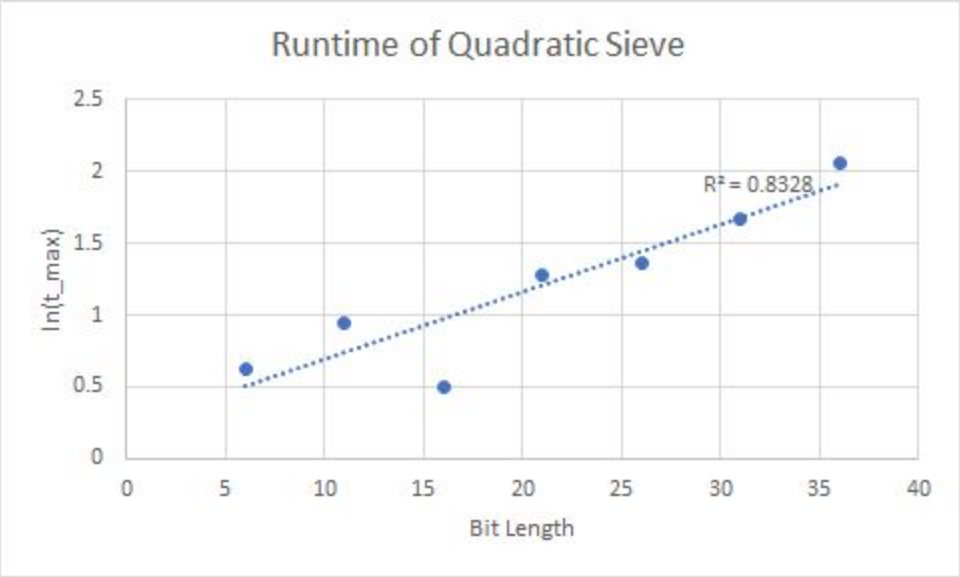

Figure 1 graphs the log of the max runtime of quadratic sieve vs the bit length of the primes. The large magnitude of R^2 being much closer to 1 than to 0 indicates a linear relationship between the two quantities. Thus, it is likely that the maximum time needed for quadratic sieve grows exponentially with prime bit length. This is consistent with modern knowledge, which states that there is no factoring algorithm that runs in sub-exponential time.

Conclusion

Looking at the results it's clear that Pollard's p-1 algorithm is only effective for small primes. Another note is that the algorithm becomes ineffective if p and q are chosen so that p-1 or q-1 factor in large prime powers. Because of this, the algorithm fails with very large primes. The algorithm also fails when prime factors of p-1 and q-1 are all the same. Our tests show that once primes get to around 16-bits, the algorithm becomes prone to failure indicating that 32-bit keys are the farthest Pollard's p-1 would be able to crack.

```
p: 29
q: 19
n: 551

Time for Pollard's p-1 algorithm to factor n: 0.0s
Pollards result: {19, 29}
Time for Quad. Sieve algorithm to factor n: 0.08676815032958984s
Quad. Sieve result: {19, 29}


p: 857
q: 983
n: 842431

Time for Pollard's p-1 algorithm to factor n: 0.0059833526611328125s
Pollards result: {857, 983}
Time for Quad. Sieve algorithm to factor n: 1.9428024291992188s
Quad. Sieve result: {857, 983}


p: 723493
q: 911527
n: 659483403811

Time for Pollard's p-1 algorithm to factor n: 0.008976459503173828s
Pollards result: {723493, 911527}
Time for Quad. Sieve algorithm to factor n: 25.305296897888184s
Quad. Sieve result: {723493, 911527}


p: 1025080536391
q: 673991833301
n: 690895910003342536156691
```

The quadratic sieve algorithm is a much more consistent and robust, harder to implement, but works better with large bit primes. However its runtime is entirely dependent on the size of what's being factored, making it extremely slow with large bit keys.

All data considered, even with more optimizations it would probably still be very difficult to factor large N's if keys were extended by even a few bits. To date, there is no known factoring algorithm that runs faster than exponential time. When taking into consideration that RSA standardly uses 1024 bit to 4096 bit keys, it makes sense why this cryptosystem is still so popular today. To give a sense of scale, a 1024-bit number would be a 309 digit number. Running in exponential time, the length to compute a factor would far exceed the life of whatever is being encrypted.

References

“Quadratic Sieve.” *Wikipedia*, Wikimedia Foundation, 26 Mar. 2019, en.wikipedia.org/wiki/Quadratic_sieve.

"Pollard's P − 1 Algorithm." Wikipedia. April 12, en.wikipedia.org/wiki/Pollard's_p_−_1_algorithm.

“Tonelli-Shanks Algorithm.” *Tonelli-Shanks Algorithm - Rosetta Code*, rosettacode.org/wiki/Tonelli-Shanks_algorithm.

Koc, Cetin K, and Sarath N. *A Fast Algorithm for Gaussian Elimination over GF(2) and Its Implementation on the GAPP*. Journal of Parallel and Distributed Computing, 1991.