An Exploration of Methods of Cracking Encryption Schemes

Alexander Agruso
Texas State University

Brandon Howell
Texas State University

Abstract

In this paper, we explore RSA and Elliptic Curve Cryptography, giving a mathematical overview of how these schemes function. Additionally, we analyze and compare various methods that exist to crack these encryption schemes. Finally, we discuss future threats to these encryption schemes and the ongoing efforts to mitigate them.

Introduction

In the age of digital communication, it is more important than ever that we safeguard our information online. With so much of our sensitive information being sent over the internet, we must find ways to ensure that unauthorized parties aren't able to view this information. The most common way of achieving this goal is through an encryption scheme.

One of the earliest known encryption schemes date back to the ancient Greeks, where Polybius invented the "Polybius Square", a primitive encryption method that encoded letters in a five by five square. Another well known encryption method from antiquity is the Julius Cipher, which shifts the letters in the alphabet so that the corresponding letters in a given sentence are scrambled in such a way as to make the text unreadable unless deciphered. [1]

While these schemes may have worked for people in the past, they are rendered essentially useless by the ubiquity of powerful computers in our modern age. As a result, much more sophisticated encryption schemes have been developed that can resist computational efforts to crack them. In this paper, we will focus on two widely used public-key encryption schemes: RSA and Elliptic Curve Encryption. These methods use advanced concepts in math, specifically number theory, to ensure that even the most powerful computers are unable to decipher a message unless they possess the private key, with which deciphering a message becomes a significantly easier task.

Both of these schemes, and all public-key encryption schemes in general, rely on intractable problems, which are problems whose solutions are difficult, if not virtually impossible to find, but are nonetheless easily verifiable once found. So, if one wants to crack these encryption schemes, one must find solutions to these intractable problems. This paper explores various methods of finding solutions to the intractable problems at the core of RSA and Elliptic Curve Encryption, as well as possible improvements that can be made upon existing methods.

RSA Encryption

The RSA encryption scheme, invented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977, is a widely used public-key encryption system that relies on the difficulty of factoring certain large integers. We provide a brief explanation of how public-private key pairs are generated, and how they are used to encrypt and decrypt messages.

The first step in creating a public-private key pair is to randomly choose two large prime numbers p and q, then calculate their product $n = pq$. Next, the value of $\lambda(n)$ is calculated, where λ is Carmichael's totient function. We define $\lambda(n)$ as the smallest integer m where $a^m \equiv 1 \pmod{n}$ for all integers a that are relatively prime to n. [2] It should be noted that $\lambda(n)$ is very difficult to calculate if only given n. However, if one has the prime factorization of n, in this case pq, the calculation becomes considerably easier, with it reducing to calculating the least common multiple of $p-1$ and $q-1$. We then choose an integer e such that $2 < e < \lambda(n)$ and e and $\lambda(n)$ are relatively prime. Finally, we find the multiplicative inverse d of $e \pmod{\lambda(n)}$. The public key consists of the modulus n and the exponent e, while the private key consists of the exponent d.

Having calculated our public-private key pair, we are now ready to start encrypting messages. Let M be the message we wish to encrypt. Using n and e, we calculate the ciphertext c by evaluating the following:

$$c \equiv M^e \pmod{n}.$$

The encrypted message c can then be sent to the recipient.

When the message is received, it can be decrypted using our private exponent d. Since d and e are multiplicative inverses, we have that $de \equiv 1 \pmod{n}$, thus we can obtain M as follows:

$$c^d \equiv (M^e)^d \equiv M^{ed} \equiv M^1 \equiv M \pmod{n},$$

thus by calculating c^d, we obtain the original message M, and since d is kept secret, only the recipient can decrypt the ciphertext.

Methods of Cracking RSA

To crack RSA encryption, we must find a way to derive the private key d from the public key values n and e. The most straightforward way of doing this is to find the prime factors p and q of n, which, as previously noted, can be used to easily calculate $\lambda(n)$, and thus calculate d.

The typical naive approach to factoring numbers is the brute force method. A custom implementation of a brute force factoring algorithm in python is given as follows:

First, we define two important utility functions, `divides(d,n)` and `isPrime(n)`, which are implemented as

```
def divides(d, n):
    q = n // d
    return n == d * q
```

and

```
def isPrime(n):
    for i in range(2, n):
        if divides(i, n):
            return false

    return true
```

respectively. The `divides` function checks whether or not d divides n, that is, it checks if n is an integer multiple of d. The `isPrime` function simply checks whether n is prime. Using these two utility functions, we can define `primefac(n)` as follows:

```
def primefac(n):
    current = n
    i = 2

    while i <= n:
        if isPrime(i) and
            divides(i, current):
            print(i)
            current = current // i

            if current == 1:
                return
        else:
            i = i + 1
```

Disallowing pre-computed tables of primes, this algorithm runs roughly in $O(2^n)$.

A more efficient, albeit far more sophisticated prime factorization method, is the General Number Field Sieve (GNFS) algorithm. Its full implementation is far beyond the scope of this paper, so we will only discuss how it compares to our brute force approach.

The time complexity of the GNFS algorithm is

$$O\left(e^{\left((64/9)^{1/3}+O(1)\right)(\log n)^{1/3}(\log\log n)^{2/3}}\right).$$

In other words, the GNFS runs in sub-exponential time. While the brute force algorithm does factor small numbers faster than the GNFS algorithm, the smaller runtime complexity of the GNFS guarantees that, past a certain point, all numbers can be factored faster by the GNFS.

Improving the Brute Force Method

Our original implementation of the brute force prime factorization algorithm is suboptimal; there are several apparent optimizations that can be made to it.

If n is prime, which is easy to check, our algorithm will still attempt to factor it, in which case it will take the longest amount of time to complete, as it will run through every number less than or equal to n before finally finding n as a factor and terminating. Additionally, given n is composite, we know that no number greater than $n/2$ can divide it, or else n would have to be a non-integer multiple of that number or prime, contrary to our assumption. Consequently, it suffices to only check factors up to $\lfloor n/2 \rfloor$. We can further reduce the number of factors we have to check by realizing that no even integer greater than 2 is prime, thus we only have to check odd numbers greater than 2 for factors. A final small optimization that we can make is not repeatedly checking whether factors are prime, and instead repeatedly dividing the same factor until it no longer divides the residue.

Implementing these optimizations, we end up with our final brute force algorithm:

```
def primefac(n):
    if isPrime(n):
        print(n)
        return

    current = n

    while divides(2, current):
        print(2)
        current = current // 2

        if current == 1:
            return

    i = 3

    while i <= n: if isPrime(i):
        while divides(i, current):
            print(i)
            current = current // i

        if current == 1:
            return

        i = i + 2
```

Despite the several optimizations made to this algorithm, it still fails to run better than exponential time. This ultimately illustrates that different, more clever approaches are needed if we wish to factor large numbers faster.

Elliptic Curve Encryption

Elliptic curve cryptography, henceforth referred to as ECC, describes several cryptographic schemes that rely on the intractability of the elliptic curve discrete logarithm problem (ECDLP). Before we can discuss how public-private key pairs are generated and how they are used to encrypt and decrypt messages, we will briefly introduce the mathematics behind elliptic curves and the ECDLP.

An elliptic curve is a cubic polynomial in two variables that takes the form of

$$y^2 = x^3 + ax^2 + bx + c,$$

combined with a "point at infinity", which we will denote as O. For the purposes of ECC, we take this curve over a finite field $\mathbb{F}_p$, where p is a carefully chosen prime number. In abstract algebra, a field simply represents a set where the usual notions of addition and multiplication hold true, and in our case, this field has finitely many elements, namely p elements.

Given the set of points on this curve over $\mathbb{F}_p$, along with O, we can endow it with a group structure by equipping it with a special "addition" operation. A group is a mathematical structure that describes the symmetries of a certain object, in this case, the points on our curve. Additionally, it allows us to define the "discrete logarithm" of points on the curve. Given two elements a and b in our group, we define the discrete logarithm $\log_b(a)$ as the integer k such that $b^k = a$. In this case, a and b are points on our curve. It should also be noted that b^k does not represent typical exponentiation; in this case, it represents

$$\underbrace{b+b+\cdots+b}_{k \text{ times}},$$

with $+$ being our special addition operation. Despite the notations being similar, the discrete logarithm and regular logarithms should not be confused with one another.

Future Threats to RSA and ECC

The invention of the GNFS algorithm instilled doubts in the security of RSA encryption. While, for the time being, large RSA keys are safe of being factored by the GNFS, it is possible that future advancements, both in computing power, and in the GNFS algorithm itself, will allow these keys to be broken relatively quickly.

Another threat, completely distinct from the GNFS algorithm, is Shor's algorithm. Invented by computer scientist Peter Shor, it is a quantum algorithm that can be used to factor prime numbers extremely efficiently. [3] It is significantly faster than even the GNFS algorithm, having a time complexity of

$$O\left((\log n)^2(\log\log n)(\log\log\log n)\right)$$

using fast multiplication. [4] For very large numbers, this runtime can be improved even further to

$$O\left((\log n)^2(\log\log n)\right)$$

using the asymptotically fastest multiplication algorithm currently known. [5] The major barrier that prevents this algorithm from being used is the lack of sufficiently powerful quantum computers to run it. In 2001, IBM successfully factored 15 into 3 and 5 using Shor's algorithm. [6] Additionally, an attempt was made by IBM in 2019 to factor 35 using Shor's algorithm, but it was unsuccessful. [7]

Future Work

References

[1] John Dooley. *History of cryptography and cryptanalysis: codes, ciphers, and their algorithms*. Springer, 2018.

[2] Wolfram Alpha. Carmichael function. https://mathworld.wolfram.com/CarmichaelFunction.html.

[3] Alice Flarend and Bob Hilborn. 213RSA Encryption and the Shor Factoring Algorithm. In *Quantum Computing: From Alice to Bob*. Oxford University Press, 04 2022.

[4] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. *Phys. Rev. A*, 54:1034–1063, Aug 1996.

[5] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n\log n)$. *Annals of Mathematics*, 193(2):563 – 617, 2021.

[6] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni,

Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. 414(6866):883–887, December 2001.

[7] Mirko Amico, Zain H. Saleem, and Muir Kumph. Experimental study of shor's factoring algorithm using the ibm q experience. *Phys. Rev. A*, 100:012305, Jul 2019.