The evolution of cryptography from Julius Caesar's times to the present

Rahul Kumar,
Research Student,
Centre for Fundamental Research and Creative Education, Bangalore, India
email:rahulkumar@cfrce.in

Abstract

The purpose of this research is to compare the strengths and weaknesses of different ciphers used through the years, as well as their difficulty to solve using modern methods. In order to do this, a program to break each cipher has been analyzed. Simple ciphers that are easier to break can be broken with simpler programs and in a smaller amount of time, while more difficult, more advanced ciphers require more complex computer programs which take more time to create and run. This research attempts to prove that the more modern a particular cipher is, the harder it is to break using a computer.

Introduction

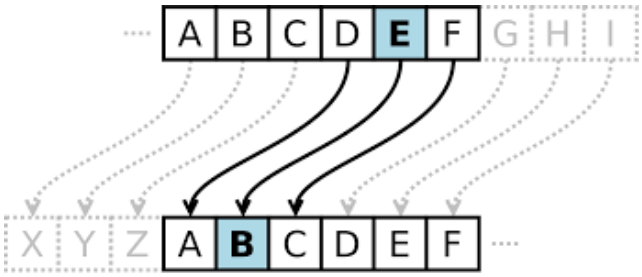
Cryptography has been a crucial factor in ensuring that secure communication has existed through the ages, from the time of Julius Caesar to the present. In Julius Caesar's time, cryptography only existed for man to man communication and could only be used by the powerful, while cryptography now has become more machine oriented and is used by almost all people today, for uses as mundane as buying groceries. Cryptography, in essence, is a race between those who make cryptosystems and those who try to break them: if a cryptosystem can be solved, it becomes unusable. As the years have passed, the strength of cryptosystems has improved as well: the Caesar cipher can be solved on paper today, while it would take our fastest supercomputers hundreds of years to crack RSA, the cryptosystem currently in use today. A cryptosystem like Enigma could not have been cracked in the 19th century, took the Allies years of dedicated work to crack it in the 20th century, and yet can be cracked with ease with computers today in the 21th century. Furthermore, RSA could be cracked in a few decades when more powerful quantum computers are built. In this paper, I will analyze the relative strengths of different cryptosystems, such as the Caesar cipher, substitution ciphers, the Vigenère cipher, the Hill cipher, Enigma, and finally El Gamal and RSA.

Study of Caesar Cipher

The Caesar cipher was one of the first ciphers recorded to have been used, as it is over 2000 years old. One of the first recorded uses of the Caesar cipher was by the Roman general Julius Caesar, after whom it is named. At the time, the Caesar cipher worked well, as frequency analysis had not yet been invented. Furthermore, most people at that time were either illiterate or only fluent in one language, so many assumed that the cipher was simply written in a language that they did not know.

Formula for encrypting: (p + s) mod 26

Formula for decrypting (c-s) mod 26



However, the Caesar cipher can easily be solved using brute force. If only lowercase letters are used in the cipher, there are only 25 possible shifts (a shift of 0 is trivial). By simply writing out all 25 different possibilities, the correct interpretation will be found. Someone using only pencil and paper could solve a Caesar cipher in less than half an hour. The advent of computers, however, means that a Caesar cipher can be solved in seconds using modular arithmetic. Each letter in the alphabet is given a number from 1 to 26 (A is 1 and Z is 26). When encrypted, the letter is moved forward by shift amount s. Therefore, the new letter is (p(plaintext) + s) mod 26. This number can then be converted back into the alphabet. Similarly, to decrypt, the formula is (c(cipher text) – s) mod 26. Computers can run through all of the possibilities quickly, and the correct one can be picked out. Although the introduction of ASCII makes a 128 letter alphabet possible, the amount of time a computer needs to break such a cipher is still trivial.

While the Caesar cipher was a good cipher for its time, developments in cryptography have rendered it obsolete long before computers arose. In the 9th century, the Muslim mathematician and philosopher al-Kindi wrote a book about breaking cryptosystems, where he introduced frequency

analysis. With frequency analysis, long Caesar ciphers could easily be broken without the need to brute force it.

Study of Substitution Cipher

Substitution Ciphers have been around for centuries, although they became prevalent long after the Caesar Cipher was first introduced. Since every letter in a substitution cipher can be mapped to any other letters, there are far too many possibilities ($4x10^{26}$) for brute force to work. For this reason, the substitution cipher remained useful for many centuries. However, in the 9th century, an Arab philosopher and mathematician named al-Kindi came up with the idea of frequency analysis. The idea behind frequency analysis is that the frequency of letters in any alphabet vary greatly. For example, in English, the letter E is over 170 times more common than the letter Z. Since each plaintext letter is always mapped to the same ciphertext letter, the most common letter in the ciphertext most likely corresponds to the most common letter in English, and so on. After a few such letters are guessed accurately, common bigrams and trigrams such as THE, AND, OR can be found, and thus more letters will be known. With computers, it has become even easier to use frequency analysis to crack substitution ciphers.

First, a computer is used to find the frequency of every letter in the ciphertext. After finding the most common letters, the most common letters in English (E, T, and A) are substituted in. Then, common two and three letter words can be filled in. Less common letters can then be substituted in. The longer any given extract is, the more likely it is that the frequencies exactly match, and the easier it will become. As more letters are filled in, it is easier to find words, which will then give the key to decrypting yet more letters.

First, In the python code, all of the characters that we need are defined. Then, to make the key, we simply rearrange all of the characters and match to them to a set of non-rearranged characters. To encrypt the function, we simply take each letter of the plaintext and map it to the corresponding letter in the key, creating our ciphertext. Decryption works the other way, where we take each letter in the key, and map it to each letter in the ciphertext, to form the plaintext.

Study of Vigenère Cipher

The Vigenère Cipher is a newer cipher than the substitution cipher, as it was created in the 16th century, originally by an Italian named Giovan Belasso. However, the Frenchman Blaise de Vigenère popularized a better version of the cipher in the 1800s, and the cipher is named after him. It utilizes a key and a plaintext, just like the previous two ciphers, but the fact that the key is as long as the plaintext makes it more difficult to break.

First, a key is chosen for the cipher. This key can be of any length. It is then repeated, so that is as long as the plaintext. Generally, the longer this key is, the harder the cipher is to break. In fact, a one time pad, a form of encryption where the key is both random and longer than the plaintext, is impossible to break. However, this is impossible because each new key has to be completely random, and both the sender and the recipient need to have a copy of it. In the Vigenère cipher, the first letter of the ciphertext is:

(P + K) mod 26, where P is the first letter of the plaintext and K is the first letter of the key. This is why the Vigenère cipher is often considered a series of Caesar ciphers in a row.

This is much stronger than a substitution cipher, because a letter in the plaintext of a Vigenère cipher is not always represented by the same letter in the ciphertext. However, while the Vigenère cipher can still be cracked rather easily with the help of the computer. Using the index of coincidence, the key length can be figured out. (The index of coincidence is significantly higher for the true key length and its multiples than it is for any other number.) When the key length is figured out, a combination of frequency analysis and brute force can be used to figure out the key.

When that is completed, the knowledge of the key is enough for the attacker to figure out the original plaintext.

Study of Hill Cipher

The Hill Cipher is a much more recent cipher, having been invented in 1929, by a man named Lester S. Hill. Unlike the previous ciphers discussed, the Hill Cipher works with matrices, and in particular matrix multiplication. The key for a hill cipher is an nxn matrix, while the plaintext is divided into blocks with n letters each. For example, a block of text with 4 letters (after being converted to numbers) would have a 4x4 matrix as its key. After matrix multiplication is carried out, the result would be n numbers, in this case 4. Those numbers, mod 26 become the ciphertext (after being converted back into letters). Matrix multiplication makes this cipher very difficult to break, as the letters of the ciphertext have very little correlation with those of the plaintext. Frequency analysis does not work on the Hill Cipher, provided that the matrix being is used is sufficiently large. However, there are still flaws in the Hill Cipher, that can be exploited to break it.

For instance, the Hill Cipher is very vulnerable to what is known as the known plaintext attack. In this scenario, attackers, along with knowing the ciphertext, know phrases that are in the plaintext. When cracking Enigma, another cipher vulnerable to this method, the Allies used German predictability to great effect. For example, messages sent using Enigma often included phrases such as “Heil Hitler” or “Nothing to Report”. In more extreme scenarios, the Allies would coerce the Germans into including a specific phrase in their reports. In one case, the Allies sent out orders for an attack on a city using a cipher that they knew the Germans had broken, and then waited as German reports were sent using Enigma including the name of that city.

Since the Hill Cipher is linear, an attacker who knows n^2 plaintexts can crack an nxn matrix in the Hill Cipher. Once the matrix is known, the ciphertext can be decoded.

Study of El Gamal Encryption

El Gamal is an advanced cryptosystem, that relies heavily on Diffie-Hellman Key Exchange. Diffie-Hellman Key Exchange is better than other methods of sharing keys. In most systems, the sender and the recipient must meet securely to set their key. However, in Diffie-Hellman, they can establish a secret key over insecure means of communication. This means that El Gamal can be used even if a key has not already been decided upon.

Diffie-Hellman Key Exchange: Both the sender and the recipient choose a base x and a mod p. The sender chooses a secret integer y, and then computes x^y mod p, which is Y. Likewise the recipient chooses a secret integer z, and then computes x^z mod p, which is Z. The sender computes Z^y mod p, while the recipient computes Y^z mod p, which are both the same. This is the key. If the number p is sufficiently large, no computer can break it. However, it is still susceptible to a man in the middle attack, where the attacker pretends that she is the recipient to the sender and that she is the sender to the recipient.

El Gamal: El Gamal encryption begins with a key generator. The recipient(Alice) finds a cyclic group G with generator g that has order q. A number x is randomly found from the group of integers from 1 to (q-1). The number h, that is g^x, is found. H, g, q, and G are published, while x is kept secret as Alice's secret key.

The next step is the encryption of the plaintext. The sender(Bob) chooses another number y from the group of integers from 1 to (q-1). C1 is g^y. Bob then calculates the number s from h^y and g^{xy}. Bob puts his message m on another element m1 in group g. He then calculates m1 * s., which is c2. Bob sends the ciphertext (c1, c2) to Alice.

The final step is decryption. First, Alice calculates s, which is equal to $c1^x$. She then calculates m1, which is c2 * s^{-1}. (s^{-1} is the inverse of s in group G). M1 is then converted back into the plaintext m.

Study of RSA

RSA is an example of an asymmetric cryptosystem. Because of this, it is very difficult to crack RSA. Even if the attacker knows the public key, he will not be able to decrypt the message. RSA is the most widely-used cryptosystem today. It was developed in 1978 by Ron Rivest, Adi Shamir, and Leonard Adleman, after whom the cryptosystem is named. RSA is based off of the fact that is very difficult to find the factors of the product of two very large prime numbers.

The first step to forming the RSA algorithm is to form the public key. First, two prime numbers p and q are chosen. Their product n is also computed. Another number e is also needed. E must be an integer, not a factor of n, and less than the totient of n. e and n make up the public key.

Next, the private key must be generated. To do this, the totient of n is first calculated. Then, the number d, which is some number k multiplied by the totient of n divided by e, is calculated. D is the private key.

In order to encrypt the message, the message must be turned into a set of numbers, called m. The ciphertext c is equal to m^e mod n. The message is then decrypted by finding c^d.

RSA is the most secure cryptosystem known to exist today, and larger-bit versions, such as 2048-bit are used by the government for sending secure messages.

Analysis

As the years have progressed, the level of difficulty in cracking ciphers has increased, and so the difficulty in using a computer to crack it has increased as well. While a Caesar Cipher can be broken with just a few lines of code, RSA is much more difficult to break. In fact, 1024-bit RSA can only be broken on supercomputers running simultaneously (It is estimated that the NSA would take one year to crack 12 1024-bit codes), and 2048-bit RSA can't currently be broken unless quantum cryptography becomes possible. Analysis of the programs used shows the increasing complexity.

Appendix

Caesar Cipher

```
import sys
def decyprt(k, cyper)
        plaintext = ‘’
        for each in cipher:
                p = (ord(each)-k)%126
                if p < 32:
                        p+=95
                plaintext += chr(p)
        print plaintext
def main(argv):
        if len(sys.argv != 1):
                sys.exit(‘Usage:brute_caesar.py’)
        cipher = raw_input (‘Enter message: ’)
        for i in range (1,95,1):
                decrypt (i, cipher)
```

```
if _name_ == “_main_”:
        main(sys.argv[1:])
```

Substitution Cipher

```
import random

chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' + \
      'abcdefghijklmnopqrstuvwxyz' + \
      '0123456789' + \
      ':.;,?!@#$%&()+=-*/_<> []{}`~^"\'\\'

def generate_key():
   """Generate an key for our cipher"""
   shuffled = sorted(chars, key=lambda k: random.random())
   return dict(zip(chars, shuffled))

def encrypt(key, plaintext):
   """Encrypt the string and return the ciphertext"""
   return ''.join(key[l] for l in plaintext)

def decrypt(key, ciphertext):
   """Decrypt the string and return the plaintext"""
   flipped = {v: k for k, v in key.items()}
   return ''.join(flipped[l] for l in ciphertext)

def show_result(plaintext):
   """Generate a resulting cipher with elements shown"""
   key = generate_key()
   encrypted = encrypt(key, plaintext)
   decrypted = decrypt(key, encrypted)

   print 'Key: %s' % key
   print 'Plaintext: %s' % plaintext
   print 'Encrytped: %s' % encrypted
   print 'Decrytped: %s' % decrypted
```

Vigenère Cipher

```
from itertools import cycle
ALPHA = 'abcdefghijklmnopqrstuvwxyz'
def encrypt(key, plaintext):
   """Encrypt the string and return the ciphertext"""
   pairs = zip(plaintext, cycle(key))
```

```
    result = ''
    for pair in pairs:
        total = reduce(lambda x, y: ALPHA.index(x) + ALPHA.index(y), pair)
        result += ALPHA[total % 26]
    return result.lower()
def decrypt(key, ciphertext):
    """Decrypt the string and return the plaintext"""
    pairs = zip(ciphertext, cycle(key))
    result = ''
    for pair in pairs:
        total = reduce(lambda x, y: ALPHA.index(x) - ALPHA.index(y), pair)
        result += ALPHA[total % 26]
    return result
def show_result(plaintext, key):
    """Generate a resulting cipher with elements shown"""
    encrypted = encrypt(key, plaintext)
    decrypted = decrypt(key, encrypted)
    print 'Key: %s' % key
    print 'Plaintext: %s' % plaintext
    print 'Encrytped: %s' % encrypted
    print 'Decrytped: %s' % decrypted
```

Hill Cipher

```
def hill(message, key, decrypt = False):
    from math import sqrt
    n = int(sqrt(len(key)))
    if n * n != len(key):
        raise Exception("Invalid key length, should be square-root-able like")
    alpha = ' ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.?,-;|'
    print "[ALPHA LENGTH]: ", len(alpha)
    tonum = dict([(alpha[i], i * 1) for i in range(len(alpha))])
    # Pad the message with spacess if necessary
    if len(message) % n > 0:
        message += '|' * (n - (len(message) % n))
    # Construct our key matrix
    keylist = []
    for a in key:
        keylist.append(tonum[a])
    keymatrix = []
    for i in range(n):
        keymatrix.append(keylist[i * n : i * n + n])
    from numpy import matrix
    from numpy import linalg
    keymatrix = matrix(keymatrix).round().T
    if decrypt:
        determinant = linalg.det(keymatrix).round()
        print "[DETERMINANT]", determinant
        if determinant == 0:
```

```
            raise Exception("Determinant ZERO, CHANGE THE KEY!")
        elif determinant % len(alpha) == 0:
            raise Exception("Determinant divisible by ALPHA LENGTH, CHANGE THE KEY!")
        inverse = []
        keymatrix =  matrix(keymatrix.getI() * determinant).round()
        invdeterminant = 0
        for i in range(10000):
            if (determinant * i % len(alpha)) == 1:
                invdeterminant = i
                break
        print "[INVERTED DETERMINANT]", invdeterminant
        # $uper l33t stuff: http://en.wikipedia.org/wiki/Modular_multiplicative_inverse
        for row in keymatrix.getA() * invdeterminant:
            newrow = []
            for i in row:
                newrow.append( i.round() % len(alpha) )
            inverse.append(newrow)
        keymatrix = matrix(inverse)
        print "[DECIPHERING]: ", message
    else:
      print "[ENCIPHERING]: ", message
    print "[MATRIX]\n", keymatrix
    # Main loop
    from string import join
    out = ''
    for i in range(len(message) / n):
        lst = matrix( [[tonum[a]] for a in message[i * n:i * n + n]] )
        result = keymatrix * lst
        out += ''.join([alpha[int(result.A[j][0]) % len(alpha)] for j in range(len(result))])
    return out
key = "GYBNQKURPGYBNQKU"
msg = "A QUICK BROWN FOX JUMPS OVER A LAZY DOG"
cipherText = hill(msg, key)
print "[CIPHERED TEXT]: |%s|\n" % cipherText
decipherText = hill(cipherText, key, True)
if decipherText.find('|') > -1 : decipherText = decipherText[:decipherText.find('|')]
print "[DECIPHERED TEXT]: |%s|\n" % decipherText
if( msg == decipherText ):
    print "[ALGORITHM] CORRECT"
else:
    print "[ALGORITHM] INCORRECT"
```

El Gamal Encryption

Program is too long to be listed here; so the link is provided here:

https://github.com/dlitz/pycrypto/blob/master/lib/Crypto/PublicKey/ElGamal.py

RSA

```
from random import randrange, getrandbits
```

```
from itertools import repeat
from functools import reduce

def getPrime(n):
    """Get a n-bit pseudo-random prime"""
    def isProbablePrime(n, t = 7):
        """Miller-Rabin primality test"""
        def isComposite(a):
            """Check if n is composite"""
            if pow(a, d, n) == 1:
                return False
            for i in range(s):
                if pow(a, 2 ** i * d, n) == n - 1:
                    return False
            return True

        assert n > 0
        if n < 3:
            return [False, False, True][n]
        elif not n & 1:
            return False
        else:
            s, d = 0, n - 1
            while not d & 1:
                s += 1
                d >>= 1
        for _ in repeat(None, t):
            if isComposite(randrange(2, n)):
                return False
        return True

    p = getrandbits(n)
    while not isProbablePrime(p):
        p = getrandbits(n)
    return p

def inv(p, q):
    """Multiplicative inverse"""
    def xgcd(x, y):
        """Extended Euclidean Algorithm"""
        s1, s0 = 0, 1
        t1, t0 = 1, 0
        while y:
            q = x // y
            x, y = y, x % y
            s1, s0 = s0 - q * s1, s1
            t1, t0 = t0 - q * t1, t1
```

```
        return x, s0, t0

    s, t = xgcd(p, q)[0:2]
    assert s == 1
    if t < 0:
        t += q
    return t

def genRSA(p, q):
    """Generate public and private keys"""
    phi, mod = (p - 1) * (q - 1), p * q
    if mod < 65537:
        return (3, inv(3, phi), mod)
    else:
        return (65537, inv(65537, phi), mod)

def text2Int(text):
    """Convert a text string into an integer"""
    return reduce(lambda x, y : (x << 8) + y, map(ord, text))

def int2Text(number, size):
    """Convert an integer into a text string"""
    text = "".join([chr((number >> j) & 0xff)
                    for j in reversed(range(0, size << 3, 8))])
    return text.lstrip("\x00")

def int2List(number, size):
    """Convert an integer into a list of small integers"""
    return [(number >> j) & 0xff
            for j in reversed(range(0, size << 3, 8))]

def list2Int(listInt):
    """Convert a list of small integers into an integer"""
    return reduce(lambda x, y : (x << 8) + y, listInt)

def modSize(mod):
    """Return length (in bytes) of modulus"""
    modSize = len("{:02x}".format(mod)) // 2
    return modSize

def encrypt(ptext, pk, mod):
    """Encrypt message with public key"""
    size = modSize(mod)
    output = []
    while ptext:
        nbytes = min(len(ptext), size - 1)
        aux1 = text2Int(ptext[:nbytes])
```

```
        assert aux1 < mod
        aux2 = pow(aux1, pk, mod)
        output += int2List(aux2, size + 2)
        ptext = ptext[size:]
    return output

def decrypt(ctext, sk, p, q):
    """Decrypt message with private key
    using the Chinese Remainder Theorem"""
    mod = p * q
    size = modSize(mod)
    output = ""
    while ctext:
        aux3 = list2Int(ctext[:size + 2])
        assert aux3 < mod
        m1 = pow(aux3, sk % (p - 1), p)
        m2 = pow(aux3, sk % (q - 1), q)
        h = (inv(q, p) * (m1 - m2)) % p
        aux4 = m2 + h * q
        output += int2Text(aux4, size)
        ctext = ctext[size + 2:]
    return output

if __name__ == "__main__":

    from math import log10
    from time import time

    def printHexList(intList):
        """Print ciphertext in hex"""
        for index, elem in enumerate(intList):
            if index % 32 == 0:
                print()
            print("{:02x}".format(elem), end = "")
        print()

    def printLargeInteger(number):
        """Print long primes in a formatted way"""
        string = "{:02x}".format(number)
        for j in range(len(string)):
            if j % 64 == 0:
                print()
            print(string[j], end = "")
        print()

    def testCase(p, q, msg, nTimes = 1):
        """Execute test case: generate keys, encrypt message and
```

```
        decrypt resulting ciphertext"""
    print("Key size: {:0d} bits".format(round(log10(p * q) / log10(2))))
    print("Prime #1:", end = "")
    printLargeInteger(p)
    print("Prime #2:", end = "")
    printLargeInteger(q)
    print("Plaintext:", msg)
    pk, sk, mod = genRSA(p, q)
    ctext = encrypt(msg, pk, mod)
    print("Ciphertext:", end = "")
    printHexList(ctext)
    ptext = decrypt(ctext, sk, p, q)
    print("Recovered plaintext:", ptext, "\n")

# First test: RSA-129 (see http://en.wikipedia.org/wiki/RSA_numbers#RSA-129)
p1 = 3490529510847650949147849619903898133417764638493387843990820577
p2 = 32769132993266709549961988190834461413177642967992942539798288533
testCase(p1, p2, "The Magic Words are Squeamish Ossifrage", 1000)

# Second test: random primes (key size: 512 to 4096 bits)
for n in [256, 512, 1024, 2048]:
    t1 = time()
    p5 = getPrime(n)
    t2 = time()
    print("Elapsed time for {:0d}-bit prime ".format(n), end = "")
    print("generation: {:0.3f} s".format(round(t2 - t1, 3)))
    t3 = time()
    p6 = getPrime(n)
    t4 = time()
    print("Elapsed time for {:0d}-bit prime ".format(n), end = "")
    print("generation: {:0.3f} s".format(round(t4 - t3, 3)))
    testCase(p5, p6, "It's all greek to me")
```

References

"Brute Force Break Caesar Cipher in Python (Python Recipe) by Captain DeadBones ActiveState Code (Http://Code.activestate.com/Recipes/578546/)." Brute Force Break Caesar Cipher in Python « Python Recipes « ActiveState Code, code.activestate.com/recipes/578546-brute-force-break-caesar-cipher-in-python/?in.

de A Franco, Joao H. RSA Implementation in Python. 17 Feb. 2012, jhafranco.com/2012/01/29/rsa-implementation-in-python/. (used under the creative commons license: https://creativecommons.org/licenses/by-nc-sa/3.0/legalcode)

Dlitz. “Dlitz/Pycrypto.” GitHub, github.com/dlitz/pycrypto/blob/master/lib/Crypto/PublicKey/ElGamal.py.

Sackett, Dan. “Implementing a Basic Substitution Cipher in Python.” Program Everyday, programeveryday.com/post/implementing-a-basic-substitution-cipher-in-python/.

Sackett, Dan. “Implementing a Basic Vigenere Cipher in Python.” Program Everyday, programeveryday.com/post/implementing-a-basic-vigenere-cipher-in-python/.

Yadav, Rohit. “Hill Cipher.” The World's Leading Software Development Platform · GitHub, raw.githubusercontent.com/rhtyd/hacklab/master/labwork/Security/hill-cipher.py.