King Abdulaziz University

EE-305

Uses of prime numbers in cryptography

Instructor: Dr. Emad Khalaf

Name: Hesham Banafa. ID: 1742275

Date: 20/4/2020

Table of contents

Abstract....3
Introduction....3
Importance of prime numbers....4
Algorithms for finding primes....4
  Book algorithm....4
  Miller-Rabin algorithm....5
The program (RSA)....5
  Key generation....6
  Encryption....6
  Decryption....7
  Signing....7
  Cracking a public key....8
Conclusion....9
Appendix A....10
  Basic prime test....10
    (python):....10
    (Flowchart):....11
  Full code history in a git repository....11
  Cracking Algorithm:....12
Bibliography....13

Abstract

In this paper, the importance of prime numbers in public key cryptography will be discussed. The focus will be on how large prime numbers are found and used in computing other values that are used in the RSA standard such as keys and why the prime numbers should be large for security reasons. We'll also review the methods to break the encryption. Furthermore, the relation between the values offer asymmetric qualities that are not used only for encryption, but verification of authors, senders or any entities holding a key such as a company or website.

Introduction

In today's world, with millions of users are using the internet, the security of user data and messages should be secure form outsiders. Cyryptography is an essential tool for any sort of secure communication and is needed for keeping government secrets and documents safe from other entities trying to harm a nation. There are many kinds of encryption that help with this task. However, most methods require both sender and receiver to know a common password, phrase or key. This can become difficult to achieve because sending the "Secret" over a connection will give it away to anyone trying to listen in. To solve this problem, public key encryption became wildly used in almost all connections.

In essence, public key encryption, gives the freedom to send a "key" to anyone who wants to establish a secure connection without fear of insecurity. For example, The RSA encryption algorithm uses key pairs that are labeled "public" and "private". The public key is then used to encrypt a message or any sort of data, to then be only decrypted with the corresponding private key, making an asymmetric relation. The key pair can also be used for making cryptographic signatures by doing to opposite operation. The signature is encrypted with the message with the senders private key. Making it only readable by the senders public key. Having unlocked the signature with a given public key makes sure to verify the origin of the message. For example, let R be a relation between a and b using a public key, where a is the clear text and b is encrypted text. Then it is true that a R b but not b R a due to the asymmetry found in the system.

A common example of uses of asymmetric encryption is whenever a user connects to a

website, both parties exchange public keys to then use this method to encrypt a temporary password that is used for symmetric encryption for transferring a lot of data. That is because asymmetric encryption is usually slower than symmetric encryption such as AES.

Importance of prime numbers

Prime numbers are essential in the RSA algorithm due to their nature. Since prime numbers have only two divisors, they can be used to make a key pair that can only be broken using prime factorization, which can become vary difficult with large numbers. Making prime numbers perfect for this application.

In the RSA system, two large prime numbers are randomly generated and used to make other calculations to produce the key pair. For example,

Let p = 11 and q = 13

Let n = p*q = 143

Then the set factors of n is {1, 11, 13, 143}

This property is used with numbers larger than 2^{4096}, which makes calculating p or q near impossible with today's computers.

Algorithms for finding primes

This section will go over common algorithms used to find prime numbers which vary in efficiency.

Book algorithm

The algorithm found in the book check for primality using a simple process.

The algorithm goes as follows for checking x:

1. If 2 divides x, then the x is not prime. Otherwise continue.
2. Let K be the largest integer less than or equal the square root of x
3. Let 1 < D <= K. Check if for all D, if D divides x, then x is not prime. Otherwise x is prime.

This algorithm is simple and effective for values less then 2^{64} due to the nature of it counting up to the square root of the number in question. For larger values, it is recommended to use a different way. The flowchart and python code are in Appendix A.

Miller-Rabin algorithm

The Miller-Rabin method try's to find if, a given n, is not prime rather than 'is prime'. In other words, it find if the number is composite and is probabilistic. Under our tests, it holds vary well. The algorithm is as follows (Miller–Rabin primality test, 1967):

1. Given n is odd integer and n > 3
2. Let K = 10 (number of rounds)
3. Choose a; 1 < a < n -1
4. Write $n - 1 = 2^s * d$ where d is odd (factor powers of 2 of n-1)
5. Evaluate the sequence $a^d, (a^d)^2, (a^d)^4, \ldots, (a^d)^{2^s} = a^{n-1}$ in mod n if a term in the sequence evaluates to 1, then n is a composite and we say 'a' is a 'witness for the compositeness'. Otherwise, continue with another 'a'.
6. If the 'last' term evaluates to -1 (K times), then n is probably prime.
7. Repeat from step 5 K times each with a random 'a'

The Miller-Rabin algorithm was tested and resulted in a much more efficient generation of prime numbers. It allows generation of primes with a size of 2^{4096} or more in seconds on a modern computer. It was chosen to be used in our program as a tool to generate prime numbers.

The program (RSA)

The program is made as an application of the RSA method and it is named after the three researchers who published their paper and thus, named after their names, (Rivest, Shamir and Adleman, 1977). In this section, the process of generating, encrypting, decrypting and signing is shown using the program, in python, made for the assignment. The Program code history link is in Appendix A. In general, the programs commands are as follows:

```
./rsa.py gen <keysize> <keyname>
./rsa.py encrypt <message> <key> <signer>
./rsa.py decrypt "<cipher>" <key>
./rsa.py export <key>
./rsa.py crack <key>
./rsa.py print <key>
./rsa.py list
```

Key generation

The program generates a random number of a given bit length from user input. The random number is then tested with the Miller-Rabin test is used to verify it is a prime number. This step is done twice to find two large primes p, q. Then n is the product of those primes. Making the only factors {1, p, q, n}. From (RFC 8017 - PKCS #1, 2016), the required calculation to generate a key pair are as follows:

n is the public key.

Let e=65537 as it is the recommended value. Part of the public key.

Then calculate phi = (p-1)*(q-1). Should be kept secret.

The private key is $d = e^{-1} \pmod{phi}$.

Now (n, e) are published to whomever is to send you encrypted messages.

And (d) is kept private to decrypt incoming messages.

Example output of the program generating a 256-bit key pair:

```
~/Development/hesham-rsa  master •  ./rsa.py gen 256 temp2
trying:  323944464035449075247891597632999626349
prime:  323944464035449075247891597632999626349
trying:  247107903300645595210864693729552181561
prime:  247107903300645595210864693729552181561
e:  65537
n:  80049237293651214852997836376731343943710039563054137452993097104210807550789
d:  1441294230839196690824075666029006299539252710544383943167836827428184716673
```

The key can then be exported as public using `./rsa.py export temp2`

Encryption

Due to the nature of RSA, the text needs to be in a numeric form. The program, thus, encodes the text, inputted from the user, into utf-8 bytes that are turned into an integer representing the word. Example output of the program converting the word “test” into an integer:

```
>> Word: test or 1953719668
```

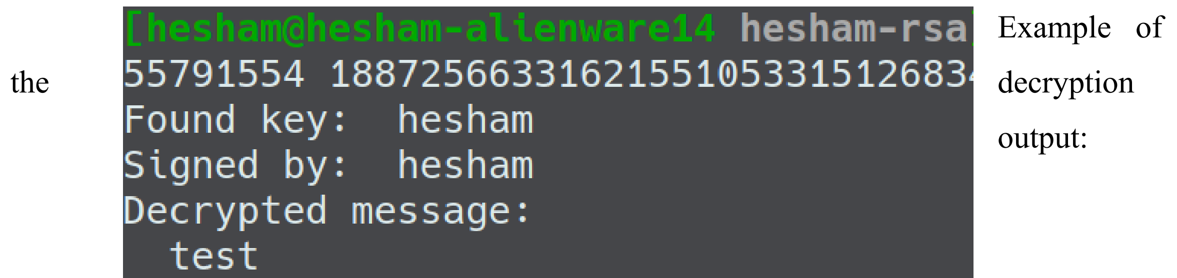
The number is then encrypted using the formula: $c = m^e \pmod{n}$ where m is the message and c is the cipher. Due to a design limitation, the numeric word can not be greater than n. Thus, each word in the sentence is split into a linked-list, and each word is processed.

```
[hesham@hesham-alienware14 hesham-rsa]$ ./rsa.py encrypt "test" hesham hesham
using n: 38127325765322279095264090686469656886117181670065112359950387633495423369249, e: 65537
Word: test or 1953719668
Encrypted msg:
  32585705105582579211558814566788684358935646946715644192110524313313555791554
```

Example of the encrypted word “test”:

Decryption

The decryption process is the reverse of the encryption. However, the private exponent, d, is needed for the calculation. Thus, we need both the modulus n and private exponent d (n, d) for the formula $m = c^d \pmod{n}$ where m is the decrypted message and c is the cipher. For this example, the message is encrypted locally where the private key is available. Example of the decryption output:



Signing

Having established a key pair having an asymmetric relation, it can be used also for authentication. In the program, any encrypted message has a reserved last “word” or “block” for a signature. The signature is encrypted, unlike the text, using the senders private part. The signeture then can only be decrypted using the corresponding public key.

For example, Hesham and Mustafa want to communicate using RSA encryption. Both have a key pair of their own. Let’s say Hesham wants to send Mustafa a message saying “Hello Mustafa”. Both words are encrypted to Mustafa using Mustafa’s public key. Then the program uses Hesham’s private key to encrypt the signature “sig:hesham” and adds it to the message as the last word.

Cracking a public key

We know that the public key as we established is the pair of numbers (n, e). We also established that the secret key “d” is calculated using p and q. Thus, the only way to find “d” is to find p **or** q. We can use prime factorization to find p or q simply with the following algorithm:

1. Let n be a multiple of p and q.
2. Generate a random number P with 1/2 bit length of n
3. Check if P is prime (Miller-Rabin), If not go to step 2.
4. If P divides n, continue.
5. Calculate q = n/P
6. Calculate phi = (P-1)*(q-1)
7. Calculate $d = e^{-1} \pmod{n}$

After step 7, we should have a “d” that can decrypt any message encrypted with (n, e). This method is possible with small key sizes. With larger keys, this method will take years to find p or q in a 2048-bit key. Which is the point of the system. An example of cracking a 32-bit key, which is vary weak:

The original key:

```
[hesham@hesham-alienware14 hesham-rsa]$ ./rsa.py gen 32 crack-test
trying:  48731
prime:  48731
trying:  56599
prime:  56599
----------------------------------------------
ID: crack-test
32-BIT KEY
PUBLIC PART:
0xa465a92d/0x10001
PTIVATE PART:
0x94df1f29
----------------------------------------------
```

Running the comand `./rsa.py crack crack-test:`

```
[hesham@hesham-alienware14 hesham-rsa]$ ./rsa.py crack crack-test
in crack
n:  2758125869
bits:  16
Trying prime:  48731
Found a factor
p:  48731
(2758125869, 65537, 2497650473, 48731, 56599, 2758020540, 'crack-test-cracked')
----------------------------------------------
ID: crack-test-cracked
32-BIT KEY
PUBLIC PART:
0xa465a92d/0x10001
PTIVATE PART:
0x94df1f29
----------------------------------------------
```

This method introduces randomness in finding the factors. A different method is to count up from a little less then half of n up to n -1. This is slower but works well with small keys. The latter is found in the code with the name crackKey2() and is currently in use. The first method is called crackKey(). The flow chart of the second method is in Appendix A

Conclusion

Through this presentation, the importance of prime numbers in cryptography is emphasized by generating and using large prime numbers. The first iteration of the program we have shown that the supplied book algorithm for finding primes works well for numbers up to 2^{128} in bit length due to it's nature of counting up to the square root of the number in question. However, for any larger values, it takes significant amount of time to determine if a given number is prime. On the other hand, the Miller-Rabin test proves to be faster for greater values. Miller-Rabin algorithm can test values between 2^{512} and 2^{4096} in seconds using a modern computer. Using this method, we can generate large keys that are based on two prime factors, thus, making communication more secure. For this reason, this method is used on our devices everytime we try to make a secure connection to the internet. Thus, we conclude that the backbone of public key cryptography are simply two rather large prime numbers.

Appendix A

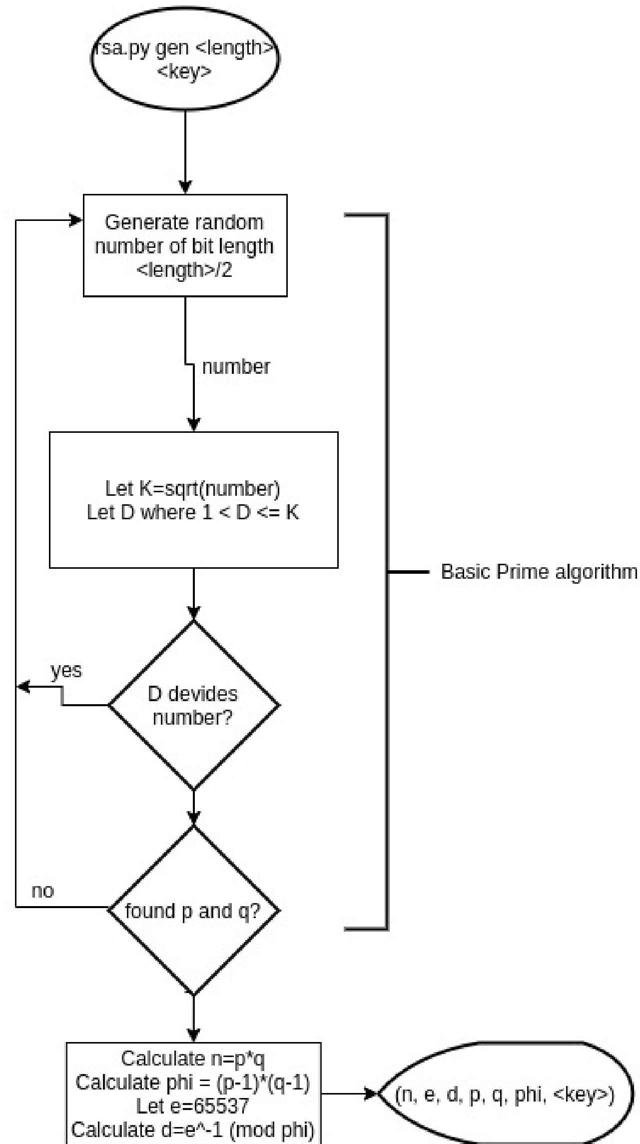
Basic prime test

(python):

```
def isPrime(number):
```

```
if number == 2:
    return True
#if 2 devides number then num is not prime. pg.21
if number % 2 == 0 or number == 1:
    return False
#largest integer less than or equal square root of number (K)
rootOfNum = math.sqrt(number)
K = math.floor(rootOfNum)
#Take odd D such that 1 < D <= K
#If D devides number then number is not prime. otherwise prime.
for D in range(1, K, 2):
    if D % 2 == 0 or D == 1:
        pass
    else:
        if number % D == 0 or number % 5 == 0:
            return False
return True
```

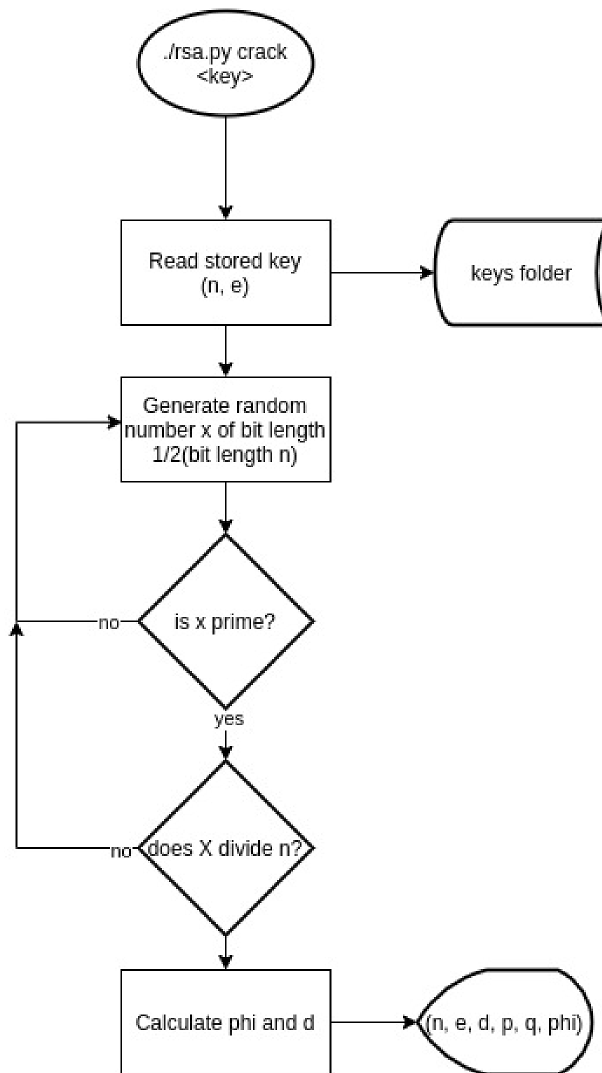
(Flowchart):



Full code history in a git repository

Code base.

Cracking Algorithm:



Bibliography

En.wikipedia.org. 1967. *Miller–Rabin Primality Test*. [online] Available at: <https://en.wikipedia.org/wiki/Miller-Rabin_primality_test> [Accessed 20 April 2020].

Tools.ietf.org. 2016. *RFC 8017 - PKCS #1: RSA Cryptography Specifications Version 2.2*. [online] Available at: <https://tools.ietf.org/html/rfc8017> [Accessed 20 April 2020].

Rivest, Shamir and Adleman, 1977. A Method for Obtaining DigitalSignatures and Public-Key Cryptosystems. *MIT*, [online] Available at: <https://people.csail.mit.edu/rivest/Rsapaper.pdf> [Accessed 20 April 2020].