Selected Topics of Embedded Software Development 2

WS-2021/22

Prof. Dr. Martin Schramm

Testing and generating Prime Numbers and Safe Primes using CryptoCore

Group 2 – Team 4
Rashed Al-Lahaseh – 00821573
Vikas Gunti - 12100861

Supriya Kajla – 12100592

Srijith Krishnan – 22107597

Wannakuwa Nadeesh – 22109097

Code enhancements

```
import random

def miller_rabin(n, k):

    if n == 2:
        return True
    if n % 2 == 0:
        return False

    s = 0
    d = n - 1

    while d % 2 == 0:
        s += 1
        d //= 2

    for _ in range(k):
        a = random.randrange(2, n - 1)
        temporary = pow(a, d, n)
        if temporary == 1 or temporary == n-1:
            continue
        for _ in range(s - 1):
            temporary = pow(temporary, 2, n)
            if temporary == 1:
                return False
            elif temporary == n - 1:
                break
        else:
            return False

return True
```

Replaced with bit shifting way

```
    s = 0
    d = n-1
    while d & 1 == 0:
        d >> 1
        s += 1
```

```
def Prime_ModExp(b,e,n,prec):
    x = (1) % n
    exp = e
    for i in reversed(range(prec)):
        x = (x * x) % n
        if(Integer(exp).digits(base=2,padto=prec)[i] == 1):
            x = (b * x) % n
        c = x
    return c
```

Replacing Squiring, multiplication old method

Figure 1: Replacing default pow() functionality

Enhanced Version of MRT

```python
def miller_rabin(n, k = 20):

    s = 0
    d = n-1
    while d & 1 == 0:
        d = d >> 1
        s += 1

        for _ in range(k):
            a = random.randrange(2, n-2)
            prec = n.nbits()
            temporary = Prime_ModExp(a, d, n, prec)
            if temporary == 1 or temporary == n-1:
                continue
            for _ in range(s - 1):
                temporary = (temporary * temporary) % n
                if temporary == n - 1:
                    break
            else:
                return False
    return True

# Taken from >> CryptoCore_User_Story_2-20211123 (Replacment of Pow(a, d, n))
def Prime_ModExp(b,e,n,prec):
    x = (1) % n
    exp = e
    for i in reversed(range(prec)):
        x = (x * x) % n
        if(Integer(exp).digits(base=2,padto=prec)[i] == 1):
            x = (b * x) % n
        c = x
    return c
```

Figure 2: MRT Algorithm

Primes Generator

```python
# --------------------------- Prime Generator -----------------------------------

def generate_prime(bits):
    while True:
        random_number = Integer((random.randrange(1 << bits - 1, 1 << bits) << 1) + 1)
        if miller_rabin(random_number):
            return random_number

# ------------------------ Safe Prime Generator ---------------------------------

def generate_safe_prime(bits):
    while True:
        random_number = Integer((random.randrange(1 << bits - 1, 1 << bits) << 1) + 1)
        if safe_prime(random_number):
            return random_number

def safe_prime(n):
    if miller_rabin(n):
        d = (n-1) >> 1
        if miller_rabin(d):
            return True
    return False
```

Figure 3: Primes Generator Functionality

Test Result

```python
# -------------------------------- TEST -------------------------------------

start_prime = time.time()
print(generate_prime(512))
finish_prime = time.time()
print('Prime takes', (finish_prime - start_prime), 'seconds')
start_safe_prime = time.time()
print(generate_safe_prime(512))
finish_safe_prime = time.time()
print('Safe Prime takes', (finish_safe_prime - start_safe_prime), 'seconds')
```

Will give us

```
26726005703375920708752062488917000550732556580998681124206130513636037477829768574148660253201512780047855713317010156364466277995271545972610032664470373

Prime takes 0.08292651176452637 seconds

24878154617189058460055071254075254521668421164241981017019616527779299463925967675239879258984497061516245226857829500948357795528344295301761845515434523

Safe Prime takes 0.1661472320556406 seconds
```

Montgomery multiplication

Is a method for performing fast modular multiplication. It was introduced in 1985 by the American mathematician Peter L. Montgomery

Montgomery modular multiplication relies on a special representation of numbers called Montgomery form. The algorithm uses the Montgomery forms of a and b to efficiently compute the Montgomery form of $ab \bmod N$. The efficiency comes from avoiding expensive division operations. Classical modular multiplication reduces the double-width product ab using division by N and keeping only the remainder. This division requires quotient digit estimation and correction. The Montgomery form, in contrast, depends on a constant $R > N$ which is coprime to N, and the only division necessary in Montgomery multiplication is division by R. The constant R can be chosen so that division by R is easy, significantly improving the speed of the algorithm. In practice, R is always a power of two, since division by powers of two can be implemented by bit shifting.

For example, suppose that N = 17 and that R = 100.
The Montgomery forms of <u>3, 5, 7, and 15</u> are 300 mod 17 = 11, 500 mod 17 = 7, 700 mod 17 = 3, and 1500 mod 17 = 4.

Addition and subtraction in Montgomery form are the same as ordinary modular addition and subtraction because of the distributive law:

$$aR + bR = (a + b) R,$$
$$aR - bR = (a - b) R.$$

This is a consequence of the fact that, because gcd(R, N) = 1, multiplication by R is an isomorphism on the additive group Z/NZ.

For example, (7 + 15) mod 17 = 5, which in Montgomery form becomes (3 + 4) mod 17 = 7.

MRT in Montgomery Domain

```python
def miller_rabin(n, k = 20, isMontgomery = False):

    s = 0
    d = n-1
    prec = n.nbits()
    temporary = 0

    while d & 1 == 0:
        d = d >> 1
        s += 1

    for _ in range(k):
        a = random.randrange(2, n-1)

        if isMontgomery:
             temporary = Prime_MontExp(a, d, n, prec)
        else:
            temporary = Prime_ModExp(a, d, n, prec)

        if temporary == 1 or temporary == n-1:
            continue
        for _ in range(s - 1):
            temporary = (temporary * temporary) % n
            if temporary == n - 1:
                break
        else:
            return False
    return True
```

```
# Taken from >> CryptoCore_User_Story_2-20211123 (Replacment of Pow(a, d, n))
def Prime_ModExp(b,e,n,prec):
    x = (1) % n
    exp = e
    for i in reversed(range(prec)):
        x = (x * x) % n
        if(Integer(exp).digits(base=2,padto=prec)[i] == 1):
            x = (b * x) % n
        c = x
    return c

# Taken from >> CryptoCore_User_Story_2-20211123 (Replacment of Pow(a, d, n) and integrating the montgomery method)
def Prime_MontExp(b,e,n,prec):
    r = 2^prec % n; r2 = (r*r) % n; rinv = inverse_mod(r,n)
    x = (1 * r2 * rinv) % n
    exp = e
    for i in reversed(xrange(prec)):
        x = (x * x * rinv) % n
        if(Integer(exp).digits(base=2,padto=prec)[i] == 1):
            x = (b * x * rinv) % n
    c = x
    return(c)
```

Test Result

```
# ------------------------------- TEST -------------------------------------

start_prime = time.time()
print(generate_prime(512))
finish_prime = time.time()
print('Prime takes', (finish_prime - start_prime), 'seconds')
start_safe_prime = time.time()
print(generate_safe_prime(512))
finish_safe_prime = time.time()
print('Safe Prime takes', (finish_safe_prime - start_safe_prime), 'seconds')

```

Will give us

```
204466626762739354657685647960825630598057253125579812380780798972913290454
123028564312144708296967270995504201361321192429536808009692794068949897909
00497

Prime takes 3.300482988357544 seconds

Safe prime, does not work on bits > 256
```