REPORT

이산대수

과목명 :	컴퓨터보안
담당교수 :	정준호 교수님
제출일 :	2021 년 04 월 18 일
공과 대학	컴퓨터공학 과
학번 : 2016112154	이름: 정동구

1. 이산대수 암호화 알고리즘.

1. RSA 알고리즘.

K^x mod N = 1 의 형태와 유사하게 암호문 = (평문)^E modN 의 형태로 암호화를 한다. 즉 RSA 암호 알고리즘의 암호문은 평문을 나타내는 수를 E 제곱해서 modN 을 취한 형태이다. 복호화는 평문 = (암호문)^D modN 의 형태로 이루어진다. 수 D 와 N 이 복호화 암호화의 키이기 때문에 (D,N)이 개인 키에 해당하게 된다. RSA 의 키 쌍 생성은 네가지 단계를 거친다. 우선 의사 난수 생성기로 소수인 p 와 q 를 구하고 둘을 곱해 N 을 만든다. 이후 p-1 과 q-1 의 최소공배수 L 을 구한다. 1<E<L 이고 gcd(E,L) = 1 인 E 를 구한다. 마지막으로 1<D<L 이면서 E*D modL =1 인 D 를 구한다. 마지막의 E*DmodL 은 암호문을 복호화 하면 평문으로 돌아간다는 것을 보증하기 위해 사용된다.

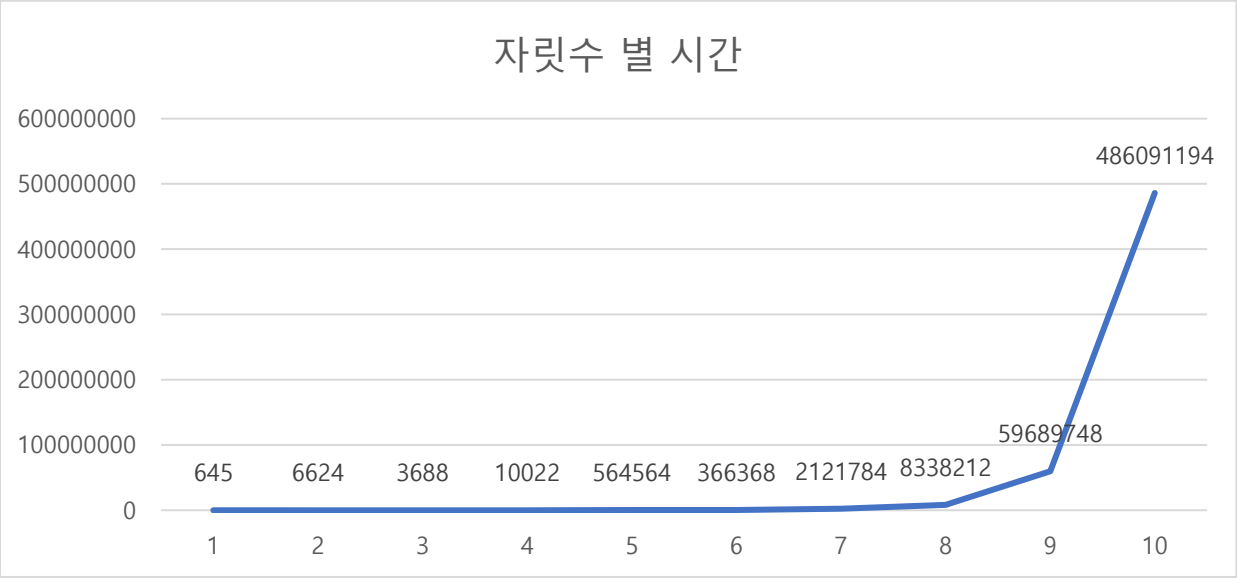
아직까지 이산대수를 구하는 빠른방법은 없기 때문에 암호문 = 평문^E 의 형태에 비해 많이 안전하다. 또한 통상 RSA 에서는 p,q 는 512 비트 이상 N 의 경우 1024 비트 이상을 이용하기 때문에 전사공격은 사실상 불가능하다. E 와 N 을 통해서 D 를 구하는 방식에는 N 을 소인수 분해하는 공격이나 p,q 를 추측하는 공격이 있지만 소인수 분해의 경우 빠르게 구하는 방식의 존재 여부나 어려운 문제인가에 대한 부분이 증명되지 않았고, 키쌍을 생성 할 때 p,q 가 기록되지 않게 주의하여야한다. 공개키 암호 모두가 가지고 있는 약점으로는 중간자 공격이 있다. 중간자 공격은 암호의 기밀성에 대한 매우 유효한 방법이다.

2. Elgamal 방식

Y = g^x modP 에서 g,y,p 값을 이용하여 지수승의 x 를 구하기 어렵다는 이산대수 문제의 어려움에 기초한다. Elgamal 방식에서는 x 가 개인키, y 가 공개키로 사용된다. Elgamal 방식의 암호화 순서는 (2~p-2)에서 개인키 X 를 선택하고 Y=a^X mod p 로 공개키 Y 를 만든다. 공개키를 전달 한 후 (0~p-1)사이의 랜덤 수 r 을 선택하고 K=Y^r mod p 를 계산한다. C1 = a^r mod p ,C2 = KM mod p 를 계산하여 C= C1||C2 를 생성한다. 생성된 암호문 C 를 전달하고 K = C1^X mod p, M=C2 / K mod P 를 계산한다.

Elgamal 암호는 같은 평문이라도 암호화가 이루어질 때마다 암호문이 달라진다는 특징이 있다. 또한 평문이 암호문이 될 때 순서쌍으로 표현되므로 암호문의 길이가 평문의 2 배가 된다. RSA 암호에 비하여 안전하지만 그만큼 더 많은 메모리 공간이 필요하며, 전송속도 또한 느리다. 따라서 RSA 암호와 비교하였을 때 속도가 느리다는 치명적인 단점을 가지고 있다.

2. 자릿수 별 X 탐색 시간



단위 : nanosec

N 을 10 진수 기준 한자리부터 10 자리까지 늘려가며 100 회씩 테스트 해본 X 를 찾는데 까지의 소요시간의 평균이다. 4 자리 이후부터 소요시간이 급격하게 증가하는 것을 볼 수 있다. 시간 측정 단위는 nanosec 를 이용하였다. 생각보다 소요시간이 짧아 자릿수가 작을 경우 micro sec 단위로도 제대로 측정이 되지 않는 경우가 발생하였다. N 의 자릿수가 10 자리인 경우에도 1 초 미만에 처리된 경우가 대다수로 속도가 빨랐다. 단 처리 속도가 기하급수적으로 증가 하기 때문에 왜 통상 RSA 에서 N 을 1024 비트 이상을 이용하는지 알 수 있다.

3. 코드

```
srand(time(NULL));

int rangeMax = 999999;
int rangeMin = 100000;
std::ofstream output("./test.txt");
clock_t start, end;
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<int> dis(rangeMin, rangeMax);
int R[100];
for (int i = 0; i < 100; i++)
{
    R[i] = dis(gen);
    std::cout << i<<" "<<R[i] << std::endl;
}

for (int i = 0; i < 100; i++)
{

    std::chrono::system_clock::time_point StartTime = std::chrono::system_clock::now();
    int k = randPrime(R[i]);
    int x = 1;
    for (int result = k % R[i]; result != 1; ++x)
    {
        result = pow(result, x, R[i]);

    }

    std::chrono::system_clock::time_point EndTime = std::chrono::system_clock::now();
    std::chrono::nanoseconds mill = std::chrono::duration_cast<std::chrono::nanoseconds>(EndTime - StartTime);
    std::cout << "<" << i << ">" << k << "^" << x << " mod " << R[i] << " = 1 | time : " << mill.count() << "nanosec₩n";
    output << mill.count() << "₩n";
}
```

```
long long pow(long long k, int x, int N) {
    if (x == 0) return 1;
    if (x == 1) return k;
    if (x % 2 == 1) return (k * pow(k * k % N, x / 2, N)) % N;
    if (x % 2 == 0) return pow(k * k % N, x / 2, N) % N;
}
```

Gdb 와 randPrime 함수는 예제를 참고하였다. 코드 대부분을 예제를 참고하여 작성하였다. 하지만 그대로 할 경우 자릿수가 늘어 날 때 거듭제곱을 하는 과정에서 시간이 급격히 많이 소모 되기 때문에 해당 부분에 분할정복 알고리즘 중 거듭제곱 알고리즘을 적용하여 시간을 대폭 단축하였다.

4. 소감

핵심이 되는 이산대수 알고리즘은 예제가 너무 잘 짜여 있어서 활용하는데 어렵지는 않았다. 문제는 자릿수가 5 이상일 때부터 몇 개는 1000 nanosec 가까이 나오다가 갑자기 연산이 길어지는 경우가 발생하였다. 코드에 문제가 있음을 발견하지 못하다가 뒤늦게 result 를 구하는 부분의 거듭제곱이 자릿수가 늘어날 경우 굉장히 급격히 오래걸림을 알고 해당부분을 분할정복 알고리즘의 하나인 거듭제곱 알고리즘을 활용하여 처리하였다. 다행히 해당 알고리즘을 적용 후 처리시간이 대폭 축소되어 10 자리까지 100 회 테스트를 빠른 시간 안에 끝낼 수 있었다. 아쉬운 점은 불과 몇주 전에 거듭제곱 알고리즘을 공부 했었다는 것이다. 알고 있음에도 해당 부분에 문제가 있음을 장기간동안 인지하지 못한 부분이 매우 아쉽다. 배우고 공부하는 것과 실전에 적용하는 것은 역시나 또 다른 문제라는 것을 또 한번 깨닫게 되었다.