**Information Security**

**Project2 Report**

**2020320054 컴퓨터학과 박민욱**

**Environment**

사용 환경: Docker Container (image: ubuntu:latest)

사용 언어: Python

사용 library : subprocess, matplotlib

**1번문제 code: main1.py, 2번문제 code: main2.py**

**Problem1**

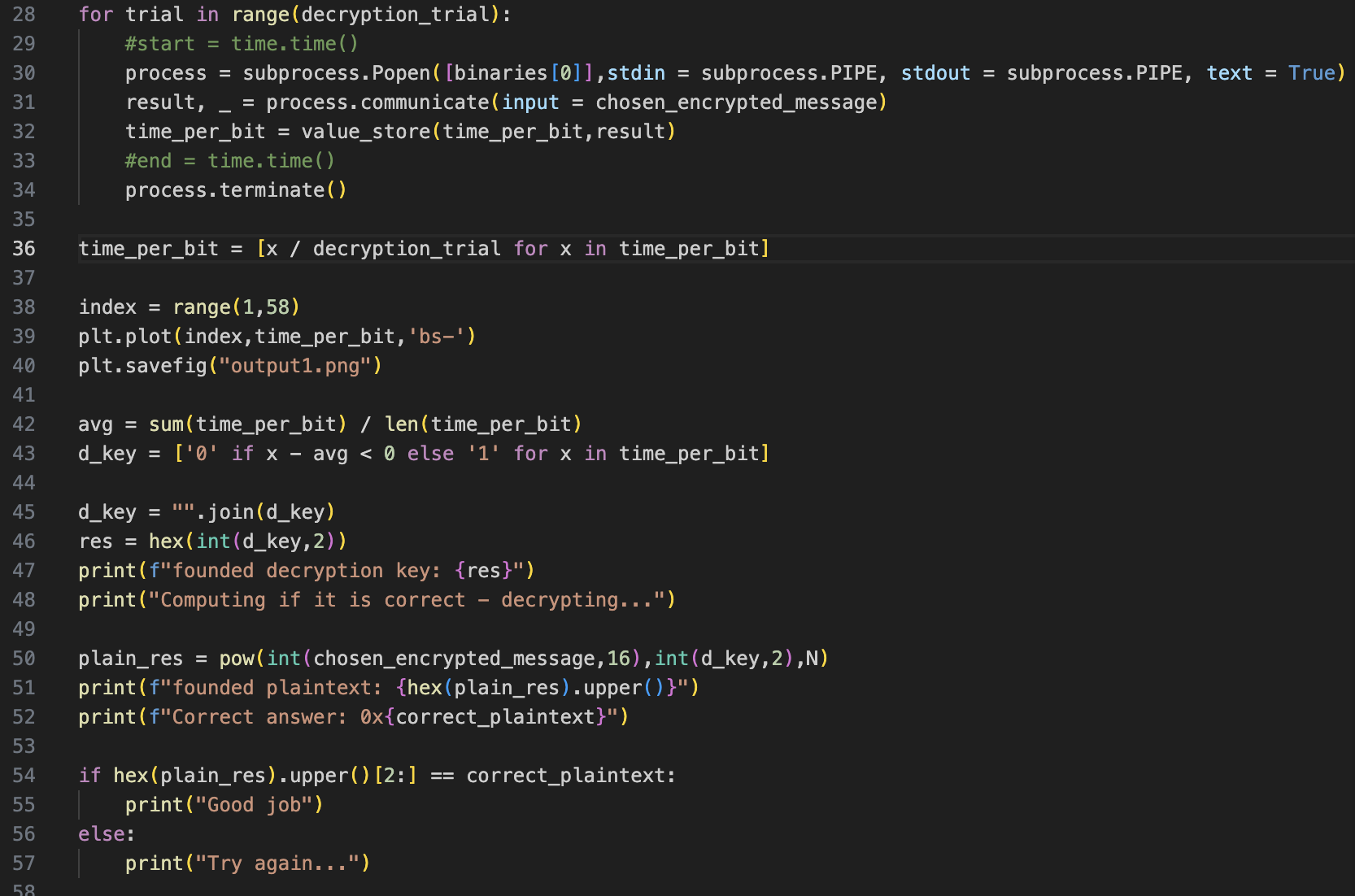
문제로 주어진 Timing attack에서 가장 중요한 부분은 decryption key에 따라 수행하는 연산의 분기가 달라진다는 점이다. 주어진 Square – Mutliply의 pseudo code를 보면, decryption key의 비트가 1인 경우 추가적인 연산을 수행하는데, problem1에서 주어진 binary는 비트 단위로 시간이 얼마나 걸렸는지를 파악할 수 있는 프로그램이다. 따라서 이 binary를 여러번 (코드에서는 100번) 실행하고 평균적인 시간이 오래걸렸던 비트는 1, 짧았던 비트는 0으로 설정하면 decryption key를 유추할 수 있다.  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Subprocess를 통해 python 코드를 스크립트로 활용하여 같은 디렉토리에 있는 바이너리 파일을 실행했고, bit당 걸린 시간을 시각화 하기 위해 matplotlib을 사용했다.

value\_store 함수의 경우는 표준 출력으로 나오는 결과를 캡쳐하여 비트 당 decryption에 걸리는 시간을 저장하는 함수이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명좌측의 함수들은 문제에서 주어진 값들과, 프로그램 실행을 위해 지정한 함수이다. Binaries 리스트에는 실행할 바이너리 경로를 작성해두었고, decryption을 진행할 message는 “ac123”으로 설정했다. 그리고 실행결과를 바탕으로 결과 검증을 위해 plaintext도 저장해 두었다. 충분한 데이터를 확보하기 위해 decryption 횟수는 100회로 지정했다. Time\_per\_bit 리스트는 각 비트당 걸리는 시간을 저장하는 변수이고, 57짜리 길이로 설정한 이유는 N이 57비트로 표현 가능해서 총 57개의 decryption time 정보가 나오기 때문이다.   


Subprocess의 Popen을 통해 서브 프로세스, 즉 바이너리 프로세스로의 입력을 PIPE, 객체로 대체하고, text=True를 통해 binary 정보가 아닌 text로 결과를 처리하게끔 했다. Communicate는 실질적인 바이너리의 실행, 실행 결과를 result에 저장하는 역할을 한다. 그리고 result는 stdout으로 나올 텍스트 파일이므로 value\_store에 이 값을 넘김으로써 time\_per\_bit에 실질적인 정보가 저장되도록 한다.   
그 다음 문단은 pyplot으로 시각 정보를 사용하는 것이다. 42,43번째 줄은 time\_per\_bit의 값을 해당 리스트의 평균으로 빼서, 음수가 나오는 경우, 시간이 적게 걸렸다는 뜻이므로 0일 확률이 높고, 아닌 경우 1일 확률이 높으므로 이를 리스트로 저장하는 역할을 한다. Join()을 통해 이 리스트를 문자열로 변환하고, pow()를 통해 구한 decryption key \*(d\_key)를 이용해서 정확한 decryption 결과와 비교하여 구한 decryption key가 정확한지 판단한다.

-결과

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명  
즉, 1번 문제의 decryption key는 아래와 같다.

**Decryption key(hex) : 0x1537f000ca50a81**

**Problem2**

두 번째 문제의 핵심은 MontMul에 있다고 보았다. Montmul에서 r >= n인 경우 r <- n-1을 시행하는데, 이 연산의 횟수가 ciphertext마다의 총 decryption time 차이를 불러일으켰다고 판단했다.

작은 크기, 중간 크기, N에 가까운 크기의 ciphertext를 총 1000개씩 뽑아서 (통계에서ciphertext의 크기로 인한 영향을 줄이기 위함) 총 3000개의 ciphertext를 활용하였다. 그 중에서 decryption time이 상대적으로 긴 ciphertext와 상대적으로 작은 ciphertext를 분류했다. 전자의 경우, r = r -n의 횟수가 decryption time 증가에 유력한 역할을 한다고 보았기 때문에, r = r – n이 발생한 비트가 1일 확률이 크다. 반면, decryption time이 상대적으로 작은 ciphertext의 경우, 높은 확률로 r = r – n이 발생하는 비트가 0이라 r = r – n 이 수행되지 않았기 때문에 시간이 짧다고 판단할 수 있다.

그러므로 각 ciphertext에서 r = r – n이 발생하는 구간을 모두 구해야하므로, Montgomery RSA decryption을 구현하고, decryption key의 비트를 모두 1로 설정했을 때의 위에서 구한 ciphertext들의 r = r – n이 발생한 비트 위치를 모두 기록한다.

그리고 0~63의 key를 가지는 dictionary를 만들고, value를 모두 0으로 초기화한다. 그리고 시간이 오래 걸린 ciphertext들의 r = r – n 발생 위치를 key로 활용하여 해당 값에 1을 더하고, 반대로 시간이 적게 걸린 ciphertext들의 r = r – n 발생 위치를 key로 활용했을 때는 1을 빼는 식으로 진행했다.   
그러면 결과적으로 value가 음수면서 절댓값이 큰 것의 key를 j라고 할 때, decryption key의 j-th 비트는 0일 것이고, value가 양수면서 절댓값이 큰 것의 key를 i라고 할 때 decryption key의 i-th 비트는 1일 것이다. 이런 방식으로 decryption key를 유추할 수 있다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명상단, 좌측 코드는 몽고메리를 직접 구현한 것이다. 다만 추가 된 점은, r = r – n이 일어난 비트를 suspicious\_bits로 추적하고 있다는 점이다. 비트를 중복해서 세면 안 되므로, set()을 활용하여 구현하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위 코드는 ciphertext 3000개를 추출하고 이에 따라 각 ciphertext의 decryption 시간을 time\_list에 저장하는 코드이다. 구현은 problem1에서 크게 벗어나지 않았다.

좌측 코드는 수집한 텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명time\_list에서 (시간이 적게 걸린 ciphertext – 걸린시간) 쌍을 short\_term dictionary에, 길게 걸린 쌍은 long\_term 딕셔너리에 저장하는 코드이다.

이전 실행으로, threshold 값은 12000, 29000이 적당하다고 판단하여 하드코딩하였다.  
두 값 모두 각각 평균적으로 200~300개의 cipher text가 추출된다.

좌측 코드는 실질적인 알고리즘을 구현한 것이다텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명. Suspicious\_bit를 모두 구해서 short\_term의 suspicious\_bit를 인덱스로 가지는 배열은 1을 빼주고, long\_term의 경우는 반대로 더해주었다.

이런 방식으로 총 길이 64의 배열이 완성되고, 이를 바탕으로 get\_ones 문자열을 완성시킨다. Decryption key의 첫번째 비트는 반드시 “0”, 마지막 비트는 반드시 “1”이므로 이는 하드 코딩해두었다.

위의 내용을 main2.py에 구현하였다.  
=> 그러나 이 방식은 실패하여 다른 방식을 시도해보았다.(계속 다른 key 값이 나왔음.)

두 번째 접근방법의 핵심은 1번 문제와 유사하게 분기 조건이다. Decryption key의 bit가 1인 경우 Multiply가 일어나고, 매우 큰 수를 곱하게 된다면 이 multiply에 걸리는 시간이 길어지기 때문에 분기 되었을 때와 그러지 않았을 때의 시간 차이가 극명하게 드러나게 된다. 그리고 우리는 현재 ciphertext를 골라서 decryption을 시킬 수 있다.   
여기서 decryption key의 MSB부터 한 비트씩 훑어가며 분기를 결정한데, 이 한 비트마다의 연산을 round라고 하자. 만일 특정 round (e.g. j-th round)에서 연산량이 극대화 되는 ciphertext를 고를 수 있다면, 이 ciphertext의 decryption 시간이 평균 이상으로 클 때 해당 round에서 bit가 1이기 때문에 극대화 된 연산량 처리로 시간이 길어졌음을 유추할 수 있다.

해당 내용은 montgomery.py에 구현해두었다. 일단 바이너리에서 시간이 오래걸렸던 ciphertext (코드에서는 “ac123”)를 골라서, Montgomery decryption을 진행할 때, 각 비트별로 걸리는 시간을 python의 time라이브러리를 이용해서 측정했다. 그리고 decryption key는 2^j 형태로 작성하여, j번째의 비트에만 분기가 일어나도록 했다.(2^1 ~ 2^62 까지 순회) 이런식으로 측정한다면, 특정 j에서 시간이 길 것이고, 이 j번째 비트가 1이었기 때문에, ciphertext의 decryption time이 길었을 것이다. 라고 추정할 수 있을 것이다.   
=> 그러나 이 방식도 decryption key당 횟수를 10000번으로 충분히 늘렸음에도 불구하고 계속해서 값이 바뀌어 실패했다.

Montgomery Algorithm의 구체적인 실행 방식에 집중하여 특정 cipher text를 구했어야했는데, 이에 실패하여 problem2의 decryption key는 구하지 못했다.