2024 암호분석경진대회 - 비밀분쇄기팀 2번 문제 풀이

복원한 비밀 키.

```
27 32 34 4B 72 43 72 79 70 74 6F 53 6F 6C 76 65
```

1 소모전력 파형 분석

문제 풀이를 위해 두 개의 소모전력 파형 데이터가 주어진다.

`2024-contest-sca-tr-reference.bin` 는 특정 평문에 대해 AES-128 암호화를 수행하는 과정 전체의 소모전력 파형이다. 해당 데이터의 파형 길이는 14000 이다. 본 풀이에서는 이를 레퍼런스 파형이라고 하겠다.

`2024-contest-sca-tr.bin` 는 AES-128 암호화를 수행하는 과정의 일부분에 대한 소모전력 파형이다. 각 파형의 길이는 850 이고, 총 1000개의 파형이 주어진다. 해당 데이터에 대응되는 1000 개의 평문-암호문 쌍이 `2024-contest-sca-pt.txt` 와 `2024-contest-sca-ct.txt` 에 주어진다. 본 풀이에서는 이를 공격 대상 파형이라고 하겠다.

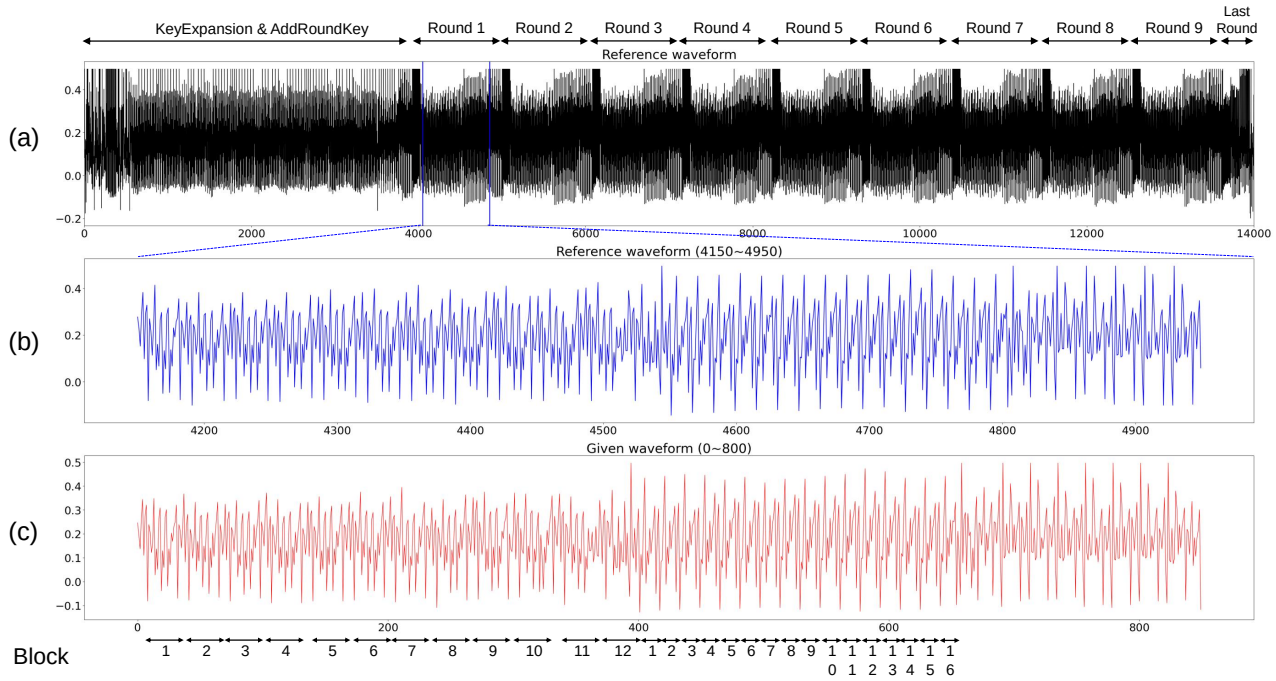

Figure 1: 레퍼런스 파형 및 공격 대상 파형 분석.

먼저 레퍼런스 파형을 분석해 보았다. Figure 1 (a) 에 보이는 것처럼, 처음 [0,4000] 구간에서 일종의 초기화 과정이 수행되고, 이후 9번의 반복되는 패턴이 있음을 알 수 있다. AES-128 의 첫 과정인 KeyExpansion 및 AddRoundKey 가 [0, 4000] 구간에서 수행되고, 이후 9번의 라운드가 반복되는 것이라고 추측할 수 있다.

주어진 공격 대상 파형이 레퍼런스 파형의 어느 부분에 해당하는 것인지를 확인해 보았다. Figure 1 (b), (c) 에 보이는 것처럼, 레퍼런스 파형의 [4150, 4950] 구간이 공격 대상 파형에 해당하는 부분이라는 것을 확인할 수 있다. 해당 부분은 AES-128 암호화의 첫번째 라운드에 해당하는 것으로 추측할 수 있다.

공격 대상 파형의 [400, 650] 구간을 확인해 보면 비슷한 패턴이 12회 및 16회 반복되는 것을 확인할 수 있다. AES 암호화의 한 라운드 내에서 같은 패턴이 반복되는 것으로는 SubBytes 및 AddRoundKeys 가 있다. SubBytes 는 128비트 데이터를 16개의 8비트 블록으로 나누어서, 각 블록의 현재 값에 따라 lookup table 에 있는 값으로 치환하는 과정이다. AddRoundKeys 는 비밀키로부터 생성된 라운드 키를 각 8비트 블록에 더해주는 과정이다.

평문-암호문 쌍들과 이에 대응되는 소모전력 파형이 주어졌을 때 가능한 공격으로 DPA (Differential Power Analysis) 가 있다 [1].

해당 공격 방식은 AES 암호화의 첫 라운드의 SubBytes 단계를 공격하는 방법으로 잘 알려져 있다. DPA 공격 및 브루트포스 서치를 적용해 비밀 키를 성공적으로 복원할 수 있었고, 그 과정을 아래에 서술한다.

2 DPA (Differential Power Analysis) 공격을 통한 비밀키의 일부분 복원

DPA 공격 개괄 DPA 는 전자 회로에 저장된 값이 바뀔 때, 새로이 저장되는 데이터 값에 따라서 소모전력 파형이 다르게 나타난다는 점에 착안한 공격 방식이다. DPA 공격은 비밀 키를 가정한 뒤 전력 모델에 따라 SubBytes 단계의 소모 전력을 예측한다. 1000개의 평문-암호문 쌍에 대해 예측한 소모 전력 값들과 주어진 공격 대상 파형 간의 연관성 (correlation) 이 높다면 비밀 키를 제대로 추측한 것이고, correlation 이 없다면 잘못된 키를 예측한 것이다.

AES-128 암호화의 키 길이는 128비트로, 비밀키 전체를 가정하고 DPA 공격을 하기에는 그 가짓수가 2^{128} 개로 너무 많다. 그러나 SubBytes 단계는 8비트 블록 단위로 치환 연산이 이루어진다는 특징이 있다. 따라서 각 8비트 블록에 대해 독립적으로 공격을 수행해서 블록 단위로 비밀키를 복원하는 방식을 사용하면 된다. 한 블록에 대한 비밀키를 복원하는 데 시도해 보아야 할 키의 개수는 $2^8 = 256$ 개이다.

특정 블록에 대한 SubBytes 단계가 공격 대상 파형에서 정확히 어떤 위치에 나타나는 지 알 수 없기 때문에, 파형의 모든 위치에 대해서 Pearson correlation 을 계산한 뒤 가장 큰 값을 취한다. 연관없는 위치라면 1000개 평문-암호문 쌍과 가정한 소모전력 간의 연관성이 없을 것이므로 correlation 이 낮게 나올 것이고, 정확한 위치에서만 큰 correlation 값이 나올 것이다.

전력 모델 DPA 공격에서 주로 사용하는 전력 모델로 Hamming Distance 과 Popcount 모델이 있다. Hamming distance 모델은 레지스터 혹은 메모리에 저장된 값이 변화할 때, 바뀐 비트 수의 개수에 따라 전력 소모가 증가한다고 가정하는 전력 모델이다. Popcount 모델은 레지스터 혹은 메모리에 저장되는 결과값의 1의 개수에 따라 전력 소모가 증가한다고 가정하는 전력 모델이다. 본 문제를 풀이할 때 Hamming Distance 및 Popcount 모델 두 가지를 모두 시도해 보았다. 결과적으로 주어진 공격 대상 파형과 높은 correlation 을 보여 성공적으로 비밀 키를 복원한 것은 Popcount 모델이다.

DPA 공격 상세 과정 공격 대상 파형의 길이를 L, 공격 대상 파형의 개수 (평문-암호문 쌍 개수) 를 N, 블록 별 시도해 보아야 할 키의 개수를 $K = 2^8$ 이라 하겠다. 파형이 저장된 행렬을 $W : [L, N]$ 이라 하자. N 개의 평문 목록을 P 라 하고, n 번째 평문의 i 번째 8비트 블록은 $P_n[i]$ 로 나타내자.

1. 128비트의 각 8비트 블록에 대해 다음 과정을 반복한다. 16개의 블록이 있으므로 총 16회 반복한다. 현재 보고 있는 블록의 번호를 i 라고 하자.
2. 행렬 $H : [K, N]$ 을 계산한다. H 의 k 행 n 열에 해당하는 값 $h_{k,n}$ 는 현재 블록에 해당하는 비밀키의 값이 k 일 때, n 번째 평문-암호문 데이터가 AES 첫 라운드 SubBytes 단계에서 소모할 것으로 예상되는 전력 값이다. 즉, $h_{n,k} = Popcount(SBox(P_n[i] \oplus k))$ 이다.
3. 행렬 $R : [K, L]$ 을 계산한다. R 의 k 행 l 열에 해당하는 값 $r_{k,l}$ 은 현재 블록에 해당하는 비밀키의 값이 k 일 때, 소모전력 추정치와 공격 대상 파형들의 l 번째 위치와의 Pearson correlation 절댓값이다.
4. 행렬 R 의 최댓값이 나타나는 위치 k_{max}, l_{max} 를 계산한다. 만약 이 값이 유의미하게 크다면, 즉 Pearson correlation 값이 유의미하게 크다면, 현재 블록의 비밀키는 k_{max} 라고 추측할 수 있고, 현재 블록의 SubBytes 단계를 수행하는 파형의 위치를 l_{max} 로 찾아낸 것이다.

DPA 공격 결과 Figure 2 에 DPA 공격 결과가 도식화되어 있다. 각 블록에 대해 가로축은 가정한 8비트 비밀키 블록의 값, 세로축은 해당 키 블록에 대해 가장 높은 correlation 값이다. 즉, 행렬 R 에 대해 행 단위로 최댓값을 계산한 결과를 도식화한 것이다. 결과를 확인해보면, 블록 0, 5, 10, 15 를 제외한 나머지 12개의 블록에서는 correlation 이 유의미하게 높은 키를 찾을 수 있었다. 즉, 128비트 비밀 키 중 96비트(8개 블록)를 성공적으로 복원한 것이다. 찾아낸 비밀키의 일부분은 아래와 같다.

```
Private key = ?? 32 34 4B 72 ?? 72 79 70 74 ?? 53 6F 6C 76 ??
```

3 Brute-force attack 을 통한 비밀키의 나머지 부분 복원

DPA 공격으로 찾아내지 못한 비밀 키의 나머지 32비트를 복원하기 위해서 brute-force attack 을 사용했다. 2^32 개의 가능한 모든 비밀 키에 대해 주어진 평문-암호문 쌍을 정확하게 암호화하는 키를 찾아내는 것이다.

Brute-force attack 결과 빠른 공격을 위해 C++ 로 소스 코드를 작성했고, OpenMP 라이브러리를 이용한 병렬화로 빠른 시간 내에 공격이 완료될 수 있도록 했다. 공격은 AMD EPYC 9654 CPU 가 2개 장착된 서버에서 수행했다. 총 384개의 논리 코어를 병렬적으로 이용해 공격을 수행했다. 코어 하나가 초당 약 10만개의 키를 시도해보고, 시스템 전체로는 초당 약 1억 9천만개의 키를 시도해볼 수 있었다. 전체 공격은 1분 이내에 완료되었다.

최종적으로 복원한 비밀 키는 다음과 같다.

```
Private key = 27 32 34 4B 72 43 72 79 70 74 6F 53 6F 6C 76 65
```

이를 이용해 주어진 1000개의 평문을 암호화하고, 주어진 암호문과 모두 같음을 확인해 검증을 완료했다.

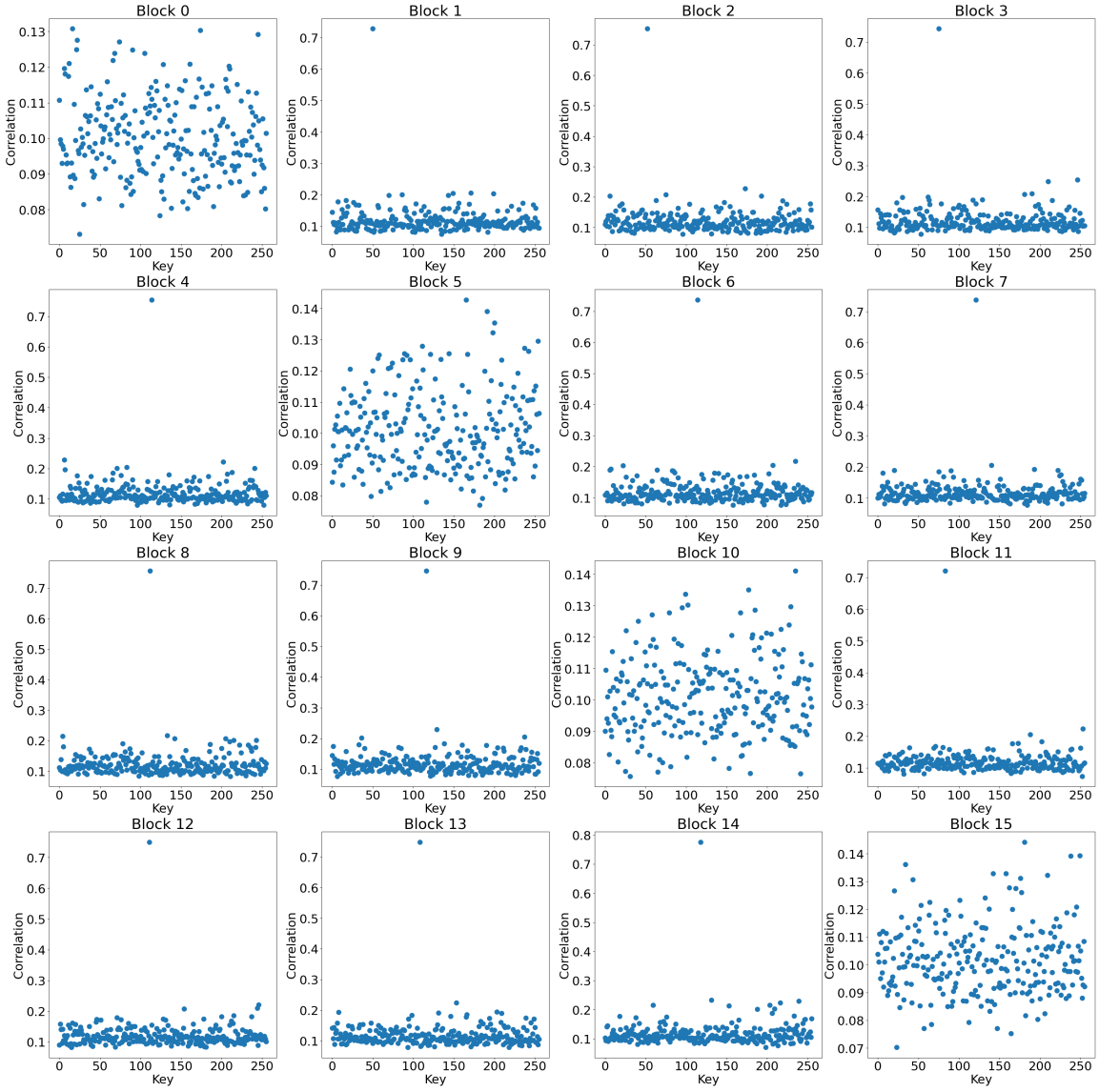

Figure 2: 16개 블록에 대한 DPA 공격 결과

4 소스 코드 설명

소모전력 파형 분석 소스 코드 `analysis.ipynb` 는 주어진 소모전력 파형 데이터를 분석하는 Jupyter notebook 소스 코드이다.

DPA attack 소스 코드 `DPA.py` 는 DPA 주어진 데이터에 대해 공격을 수행하는 소스 코드이다. 실행에 약 2분이 소요되며, 결과를 도식화한 이미지 파일 `DPA_Result.png` 가 생성된다. 다음과 같이 실행한다.

```
$ python DPA.py
Block00, Found Key: 0x10, Correlation: 0.131
Block01, Found Key: 0x32, Correlation: 0.729
Block02, Found Key: 0x34, Correlation: 0.753
Block03, Found Key: 0x4b, Correlation: 0.743
Block04, Found Key: 0x72, Correlation: 0.754
Block05, Found Key: 0xa5, Correlation: 0.143
Block06, Found Key: 0x72, Correlation: 0.737
Block07, Found Key: 0x79, Correlation: 0.739
Block08, Found Key: 0x70, Correlation: 0.756
Block09, Found Key: 0x74, Correlation: 0.745
Block10, Found Key: 0xeb, Correlation: 0.141
Block11, Found Key: 0x53, Correlation: 0.721
Block12, Found Key: 0x6f, Correlation: 0.750
Block13, Found Key: 0x6c, Correlation: 0.749
Block14, Found Key: 0x76, Correlation: 0.776
Block15, Found Key: 0xb5, Correlation: 0.144
```

Brute force attack 소스 코드 블록 0, 5, 10, 15에 대해 Brute force attack 을 수행하는 소스 코드가 bruteforce 디렉토리에 들어 있다. `AES.cpp` 및 `AES.h` 는 오픈소스 AES 구현체, `bruteforce.cpp` 는 공격을 수행하는 소스 코드이다. 다음과 같이 컴파일하고 실행한다.

```
$ g++ -O3 AES.cpp bruteforce.cpp -fopenmp -o bruteforce
$ ./bruteforce
... (many lines)
(Thread 7) throughput = 105572.68 keys / sec
(Thread 39) throughput = 105528.74 keys / sec
(Thread 336) throughput = 105474.28 keys / sec
(Thread 296) throughput = 126566.12 keys / sec
(Thread 339) throughput = 105441.42 keys / sec
(Thread 363) throughput = 105427.95 keys / sec
Found key: 27 32 34 4b 72 43 72 79 70 74 6f 53 6f 6c 76 65
```

답안 검증 소스 코드 `validate.py` 는 찾아낸 키를 이용해 주어진 1000개의 평문을 실제로 암호화해 주어진 암호문과 정확히 일치하는지 검사하는 코드이다. 다음과 같이 실행한다.

```
$ python validate.py
Key is correct!
Key: 0x2732344B7243727970746F536F6C7665
```

References

[1] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1:5–27, 2011.