2020 제 3 회 부채널분석 경진대회

한성대학교 정보컴퓨터공학과 권혁동

한성대학교 IT 융합공학과 김현지

[문제 3] 1 라운드 첫 번째 바이트 비밀키를 단일 파형으로 찾기 위한 최적의 딥러닝 기반 부채널 분석 기술 개발 및 학습 네트워크 도출

[data sets] AES-STM, Unprotected AES (variable key)

- ML-STM-AES.btr
  5000 개의 파형과 각 파형 당 500001 개의 포인트
  x 축 : 시간, y 축 : 전력 소모량
- ML-STM-AES-plain.txt
  16 bytes 의 평문 5000 개
  → labeling.py 의 parse_from_hex 함수 통해 각 평문에서 1 byte 추출
- ML-STM-AES-cipher.txt
  16 bytes 의 암호문 5000 개
- STM-AES-key.txt
  16 bytes 의 키 5000 개
  → labeling.py 의 parse_from_hex 함수 통해 각 키에서 1 byte 추출
- ml-avr-aes-label.npy
  각 평문의 1 바이트에 대해 variable key 와 XOR 연산한 후 Subbyte 한 중간값
- traces.npy
  각 평문의 1 바이트에 대해 variable key 와 XOR 연산 후 subbyte 한 값
  (중간값)을 구하는 시점의 전력 소모량 (파형)

[딥러닝 기반 부채널 분석 기술 개발 보고서]

1. 학습을 위한 주요 시점 선택

* AES 구조

1. 10 round 이며 한 round 당 subbyte - shiftrow - mix column - addroundkey 로 구성
2. 1 round 전에 key schedule 과 addroundkey 수행
3. 마지막 10 round 에서는 mix column 연산을 수행하지 않음

이와 같은 구조로 인해 그림 1 과 같이 반복되는 9 개의 round 과 마지막 10 round 로 나누어진다. AES 의 Subbyte (S-box) 연산은 비선형 연산이므로 subbyte 구간까지의 중간값은 계산 가능하며, 해당 중간값에 대한 전력 소모량을 통해 키를 찾아낼 수 있다. 따라서, 주요시점은 subbyte 구간으로 설정한다.

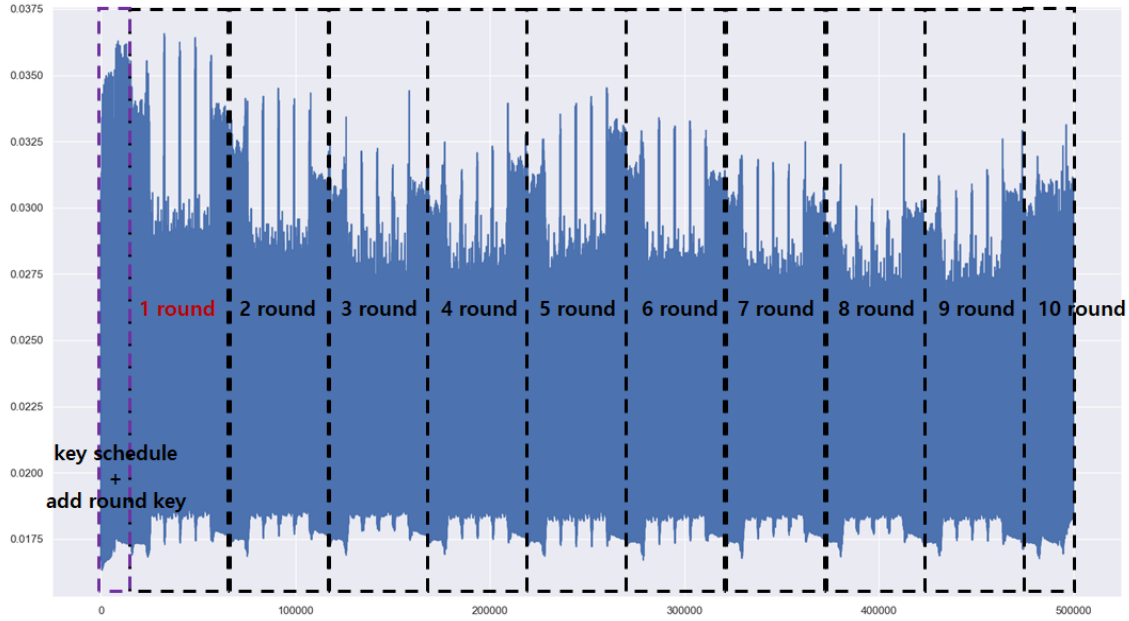


그림 1. 전체 파형

그림 2 는 subbyte 구간을 찾기 위해 한 round 의 파형을 확대하여 보여준다. subbyte 구간부터 1 라운드가 시작되며, subbyte - shiftrow - mixcolumn - addroundkey 순서로 수행된다. subbyte 연산은 대략 16200 ~ 23000 구간에 걸쳐 진행되며 그림 3 에서 자세히 살펴본다.

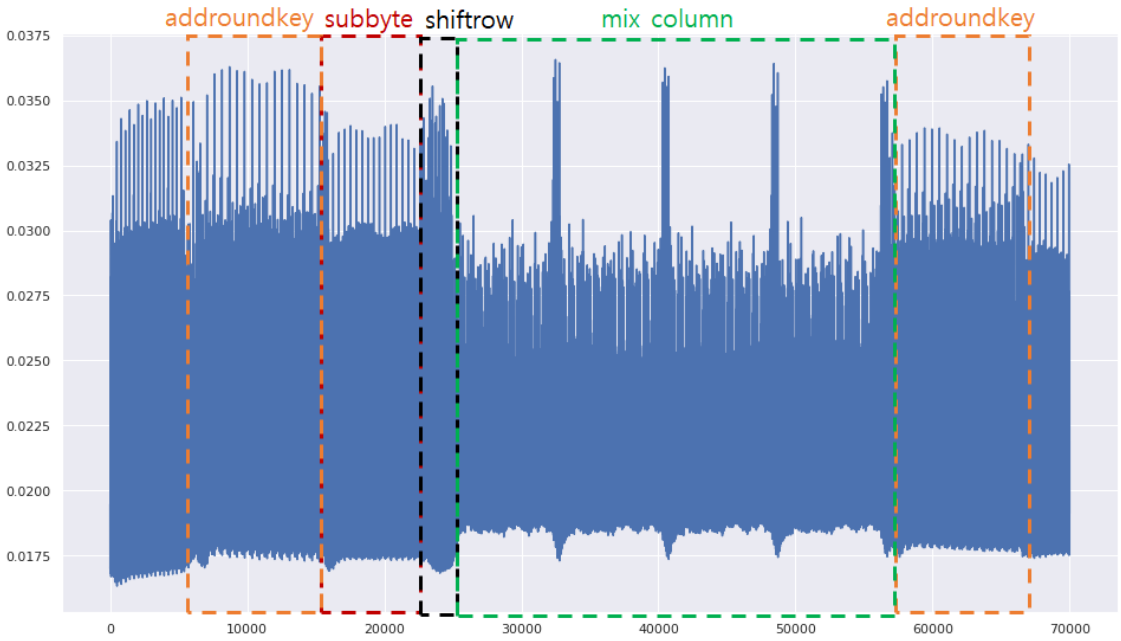


그림 2. 한 round 의 구조 (subbyte - shiftrow - mixcolumn - addroundkey)

subbyte 구간은 16 bytes 에 대한 연산을 진행하므로, 총 16 개의 피크가 나타난다. 이 때 각 byte 에 대한 중간값을 구하는 연산이 진행되었음을 유추할 수 있다. 따라서, 1 round 의 16 bytes 의 마스터 키를 찾을 수 있는 최소 파형수는 대략 16200 ~ 22600 (총 6400 points) 이다.

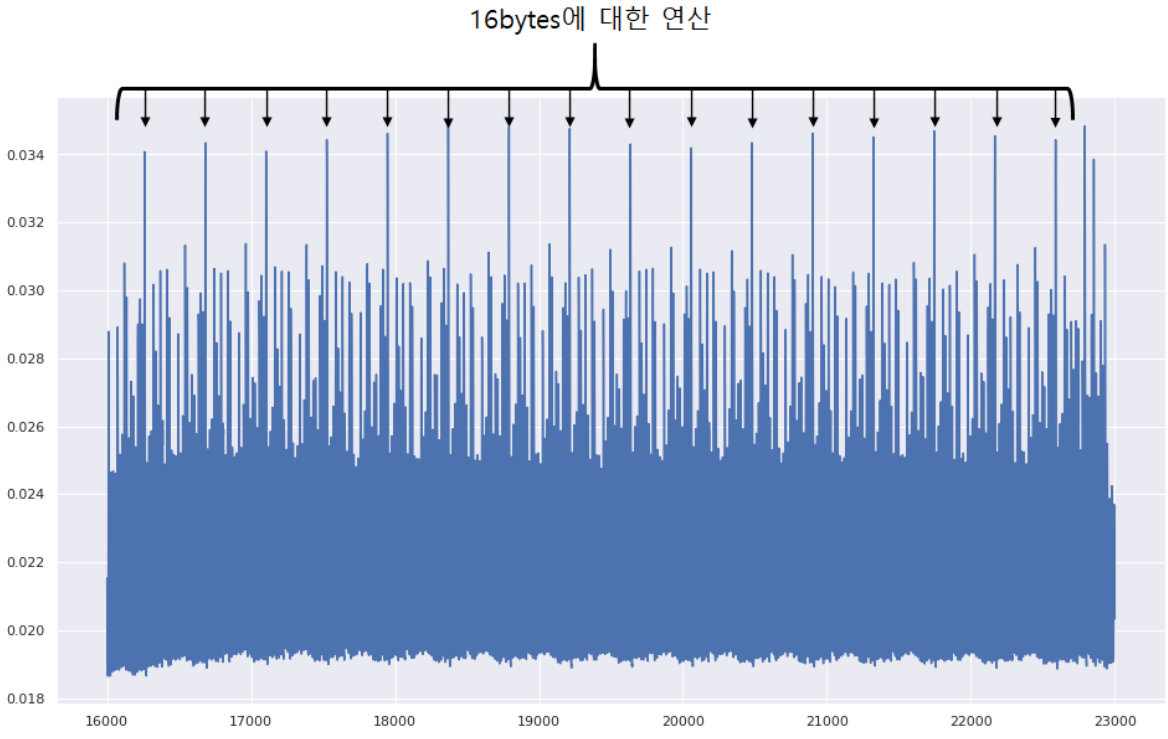


그림 3. subbyte 구간 (16200 ~ 22600)

그림 4와 같이 16번의 피크가 나타나며 이는 16번의 연산이 수행되었음을 나타낸다. 1 round의 첫 byte의 비밀키를 찾는 것이 목적이므로 16개의 구간 중 첫번째 연산 시점을 대상 파형으로 선정한다. 따라서 시점(x축)이 16200~16300인 100개의 파형 샘플을 추출하였다. 이후, 해당 구간을 범위 내에서 변경해가며 실험해보았으나, 16240~16270과 같이 너무 좁게 설정한 경우 성능이 감소하는 것을 볼 수 있었다.

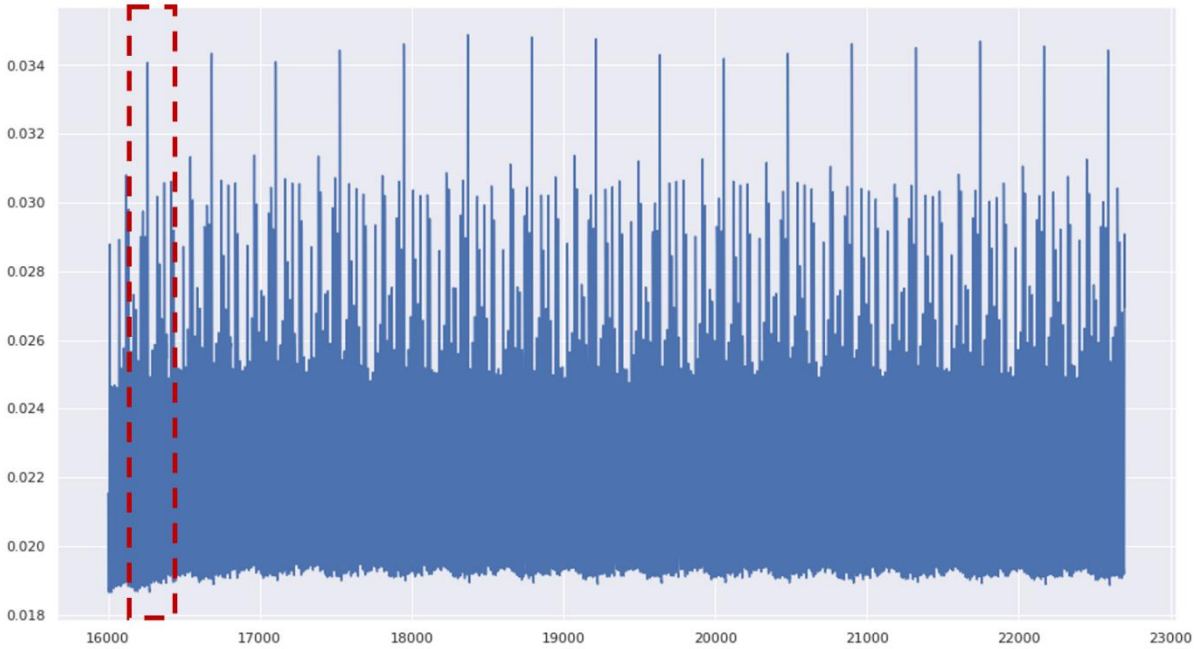


그림 4. 1라운드의 첫 바이트 시점

[학습된 네트워크에 대한 구현 소스코드]

Data set

1. traces.npy
   5000개의 파형에 대한 16200 – 16300 구간의 파형 정보
   Shape : (5000,101)
2. labeling.npy : 5000개의 평문과 가변 키를 대응시켜 XOR연산 후 S-box 이용하여 대치한 값
   shape : (5000,1)

Idea

Template attack을 사용하였다. 비밀 값을 특징화시켜 템플릿을 추정하는 프로파일링 단계와 공격 대상 장비로부터 얻은 파형의 비밀정보를 추출하는 매칭 단계로 구성된다. 이러한 특징은 인공지능과 매치된다. 프로파일링 단계는 파형과 그에 해당하는 중간값을 data와 label 형태로 구성하는 것이고, 공격 대상 장비로부터 얻은 파형의 비밀정보를 추출하는 단계는 테스트 데이터를 입력하여 추론을 통해 해당 데이터의 label을 얻어내는 과정과 유사하다.

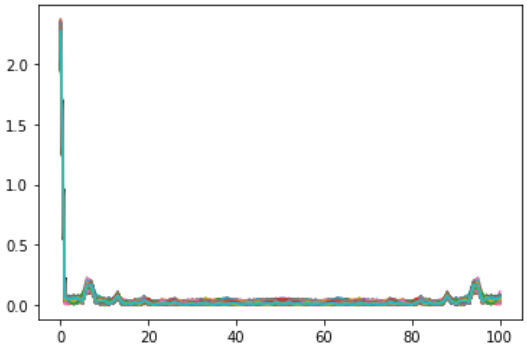
Data preprocessing

신경망을 통한 학습에 앞서 raw trace를 전처리하는 과정을 수행하였다. 수집한 소비 전력에 지나친 잡음이 존재하거나, 시점이 많은 경우 효율적이지 않으며 분석 성능이 감소할 수 있다. 특히 전력 파형의 수가 제한된 경우 잡음을 제거하는 것이 필수적이다. 따라서 전처리 과정을 통해 파형의 형태, 차원을 변환하거나 데이터를 수정해야할 필요가 있다. 성능을 향상시키기 위해 2가지 전처리 기법을 적용해보았다.

1. FFT

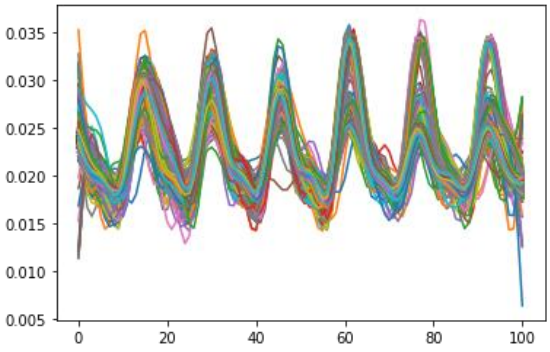
이산 푸리에 변환을 통해 전력 파형을 분리하여 특정 대역폭의 데이터를 제거한 후 역변환하여 잡음을 제거하는 방법이다. 저대역폭의 데이터만 남길 경우 Low-pass filter를 사용한다. 본 실험에서는 FFT와 Low-pass filter, 차원축소를 적용해보았다.

```
X = abs(fft(X))
```



<FFT>

```
import scipy.signal as signal
bbbb, aaaa = signal.butter(8, 0.3, 'lowpass')
X = signal.filtfilt(bbbb, aaaa, X)
```



<Low-pass filter>

2. 차원 축소

해당 데이터는 시점이 100인 100차원의 데이터이다. 5000개의 파형 수에 비해 차원이 많은 것으로 판단되어 데이터의 차원을 축소하면서 정보량은 유지하는 차원 축소를 수행하였다. 소비 전력 파형에 적용할 경우, 주요 시점을 선택하기 용이하며, 이것은 성능에 큰 영향을 미친다고 한다. 해당 실험에 사용한 Template Attack은 유의미한 시점에 대한 표본 평균과 공분산 행렬을 계산하여 진행되기 때문에, 차원 축소가 이루어질 경우 행렬의 크기가 적어져 연산시간이 단축될 수 있다.

유의미한 시점을 찾기 위해 차원 축소에는 오토인코더를 사용하였다. 해당 100차원의 정보를 dim_latent_vector의 값을 설정하여 1~10차원으로 줄여 실험한 결과 6차원이 가장 높은 검증

정확도를 보였다.

```
# Auto encoder
  inputs = Input(shape = (X_train.shape[1],))
  layer_1 = Dense(128, activation = 'relu')(inputs)
  latent_vector = Dense(dim_latent_vector, activation = 'relu')(layer_1
)
  layer_2 = Dense(128, activation = 'relu')(latent_vector)
  output = Dense(X_train.shape[1])(layer_2)
  ae = Model(inputs  = inputs, outputs = output)

  # encoder
  encoder = Model(inputs = inputs, outputs = latent_vector)

  # compile & fitting
  ae.compile('adam', 'mse')
  ae.fit(X_train_scaled, X_train_scaled, epochs = 10, validation_split=
 0.05, verbose =0)
```

[Deep learning model]

학습에는 3가지 모델을 사용하였으며, 공통적인 사항은 다음과 같다.

1. 파형과 중간값을 매치하여 학습, key 값을 찾기 위해
2. output neuron 수 : 256 (중간값의 경우의 수가 0~255이므로)
3. Loss function : categorical crossentropy (다중 클래스 분류)
4. Metrics : accuracy
5. Hidden layer activation : relu

***학습**

1. MLP

해당 모델은 단순한 전연결층 레이어를 사용하였으며, input_dim으로는 사용된 파형의 포인트의 수로 설정하였다.

```
def set_model():
  model=Sequential()
  model.add(Dense(400, input_dim=range2-
  range1, kernel_regularizer= l1_l2(0.00, 0.0001)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(Dense(200, kernel_regularizer= l1_l2(0.00, 0.0001)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(Dense(256, activation='softmax'))
```

```
  optimizer = Adam(lr=0.0001)
  model.compile(loss='categorical_crossentropy', optimizer=optimizer, m
etrics=['accuracy'])


  return model
```

2. CNN

CNN의 경우 입력 데이터가 한 차원 많기 때문에 파형의 형태를 변경해주었다.

```
def cnn(classes=256):
  input_shape = (range2-range1,1)
  img_input = Input(shape=input_shape)
  x = Conv1D(32, 3, activation='relu', padding='same', name='block1_con
v1')(img_input)
  x = AveragePooling1D(2, strides=1, name='block1_pool')(x)
  x = Conv1D(64, 3, activation='relu', padding='same', name='block2_con
v1')(x)
  x = AveragePooling1D(2, strides=1, name='block2_pool')(x)
    x = Flatten(name='flatten')(x)
  x = Dense(classes, activation='softmax', name='predictions')(x)

  inputs = img_input
  model = Model(inputs, x, name='cnn_best')
  optimizer = Adam(lr=0.0001)

  model.compile(loss='categorical_crossentropy', optimizer=optimizer, m
etrics=['accuracy'])
  return model
```

3. RNN

파형 정보가 시계열 데이터라고 생각하여 실험해보았으나, 성능의 변화가 없었다.

```
input = Input(shape = (X_train_seq.shape[1], X_train_seq.shape[2]) )
layer_1 = RNN(LSTMCell(256),return_sequences= False )(input)
layer_2 = Dense(256, activation ='softmax')(layer_1)
model = Model(inputs = input, outputs = layer_2)
```

전처리 과정을 거쳤기 때문에 복잡한 구조가 필요하지 않다고 생각되며, 성능은 모두 비슷한 수준이다. 실험 결과 정확도는 훈련하지 않을 경우, 약 0.004의 확률인데 학습을 거친 경우 약 10배정도 정확도가 상승하였다. 그러나 정규화 레이어 등을 추가하였음에도 불구하고 검증 정확도가 잘 오르지 않았으며, Variable key여서 매번 다른 평문과 다른 키값을 사용하였기 때문이라고 생각된다. 해당 MLP모델과 동일한 구조로 더 많은 수의 fixed key에 대해 실험했을 때, 90퍼센트의 정확도를 얻을 수 있었다.

***공격 수행**

model.predict를 통해 공격 파형이 0~255의 각 중간값을 갖게 될 확률을 구한다.
Probs에는 각 공격 파형 당 0~255의 중간값으로 분류될 확률이 저장된다.
해당 배열에서 최대값을 갖는 요소의 인덱스를 반환함으로써 입력으로 사용된 각 파형들이 갖는 중간값을 예측한다. 이후 반복문을 통해 각 파형이 해당 중간값을 갖기 위해 필요한 키를 구한다.
키값은 0~255의 범위 내에서 무차별 대입하며, 각 파형에 대응되는 평문과 XOR된 후 바이트 대치를 통해 중간값을 얻는다. 학습을 통해 예측된 중간값과 무차별대입하여 얻어낸 중간값이 같아질 때의 키가 추측키이며, 실제 키와 같은 경우 공격에 성공한다.
다음은 공격 수행 코드이며, 그림 5는 공격 수행 결과이다.

```
probs=model.predict(X_test)
guess_labels=np.argmax(probs,axis=1)

num_start = 0
cnt = 0
real_key = keys[4500:]

for j in range(X_test.shape[0]):

        print("========{}th attack==========".format(j))
        for k in range(256):

            pt = plaintext_test[j].item()

            label_for_key = AES_Sbox[pt ^ k]

            if (guess_labels[j] == label_for_key):
              print("key : {key}".format(key = k))
              print("real key : {real}".format(real = real_key[j]))
              if(k == real_key[j]):
                print("attack success!")
                cnt +=1

num_start = num_start+X_test.shape[0]
print ("the number of attack success : {}".format(cnt))
```

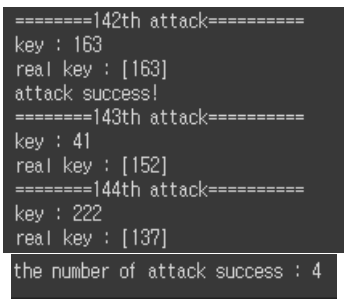
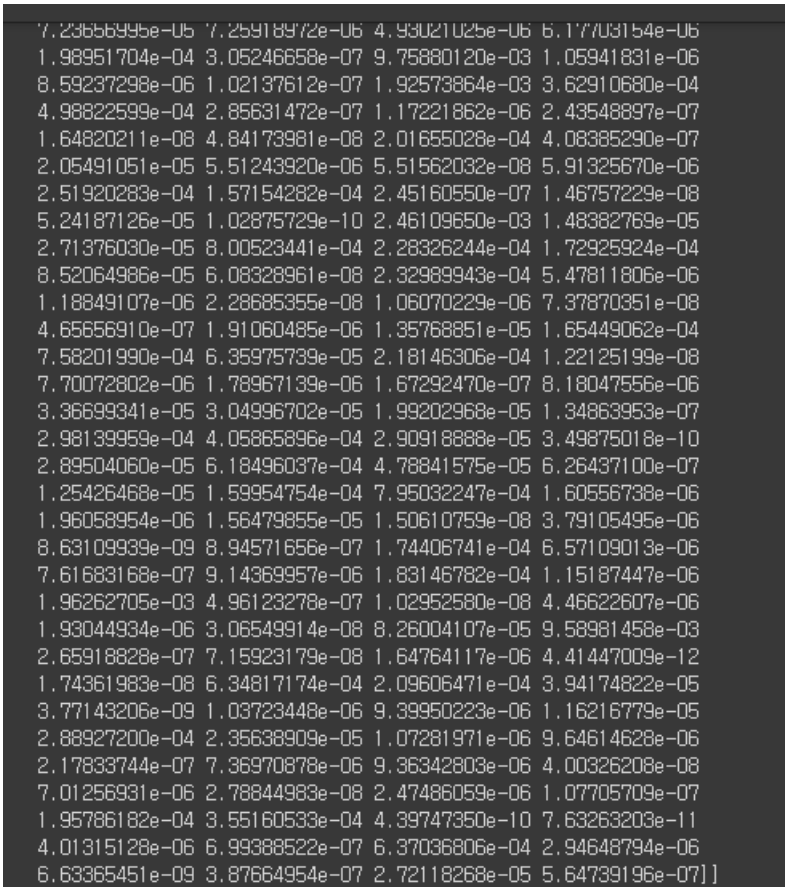


그림 5. 공격 수행 결과



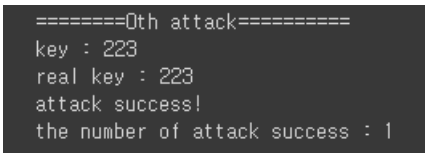


그림 6. 각 분류 확률과 키