SDES 암호 분석 실험

임세진

https://youtu.be/EOi9VN4K4M8

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

CryptoCraft LAB

01. SDES

- SDES (Simple-Data Encryption Standard)

✓ DES를 간단하게 표기한 것

✓ DES와 비트 수만 다르고 암호화 방법이 동일 (교육용 DES라고 생각하면 됨)

✓ Feistal 블록 암호 : 암호화 방식이 특정 계산 함수의 반복으로 이루어지는 암호

✓ Feistal 구조 : 데이터를 두 부분으로 나누어 좌, 우에 교대로 비선형 변환을 적용시키는 구조

	DES	S-DES
Block size	64 bit	8bit
Key size	56 bit	10 bit
Round	16	2

01. SDES

<키 생성>

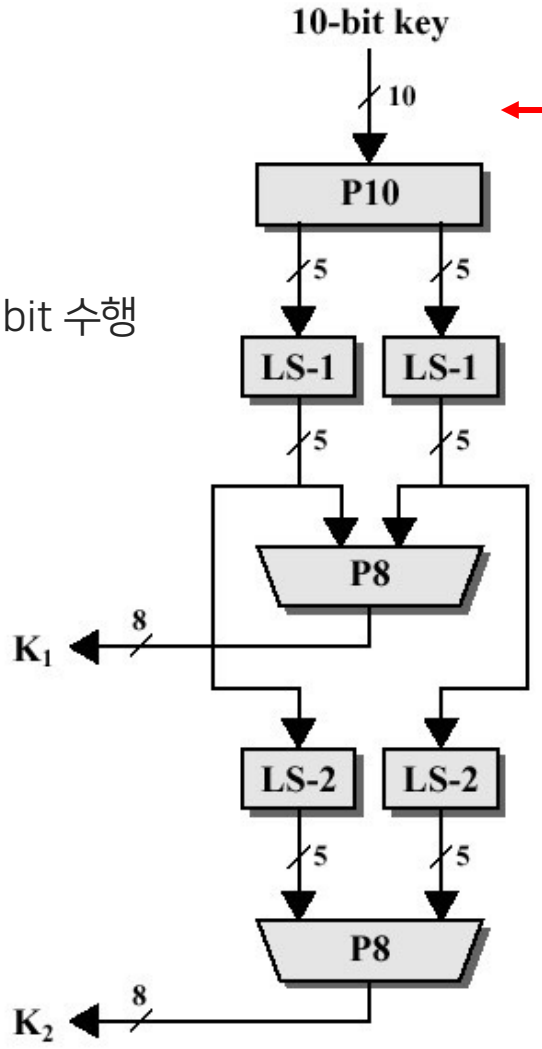
1) 10bit의 키를 P10의 순열로 치환

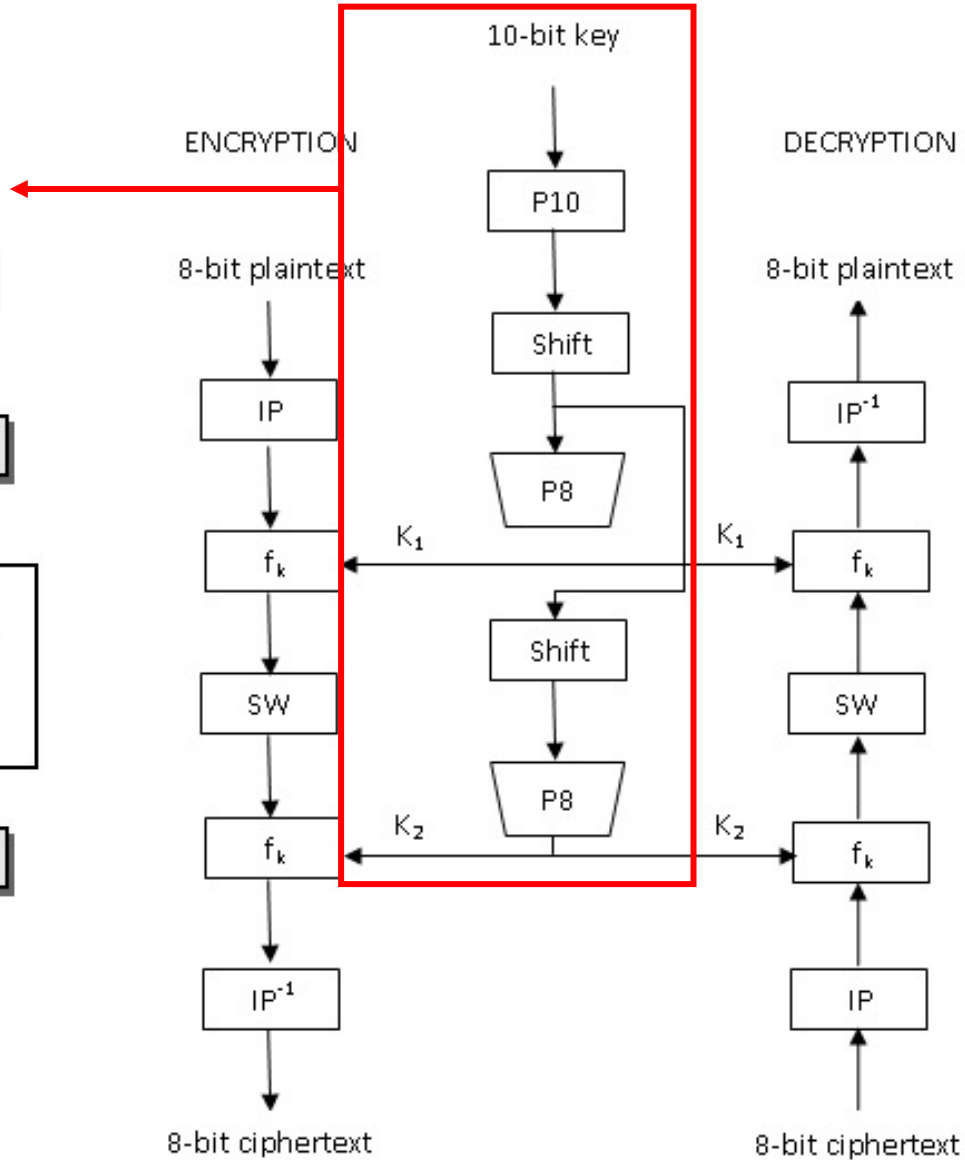
2) 재 정렬된 값을 좌우 5bit씩 나눈 후 각각 LS(Left Shift)-1bit 수행

3) 2의 결과를 합쳐서 P8의 순열로 치환하여 8bit의 K1 생성

4) 2의 결과를 다시 각각 LS(Left Shift)-2bit 수행

5) 4의 결과를 합쳐서 P8의 순열로 치환하여 8bit의 K2 생성





01. SDES

<암호화>

1) 평문 8bit → IP(Initial Permutation) 순서로 치환

2) 1의 결과를 좌우 4bit씩 L과 R로 나눔

3) R을 E/P(확장순열)에 입력하여 8bit로 확장

4) 3의 결과를 K1과 XOR

5) 4의 결과를 S-boxes에 입력하여 4bit 결과 생성

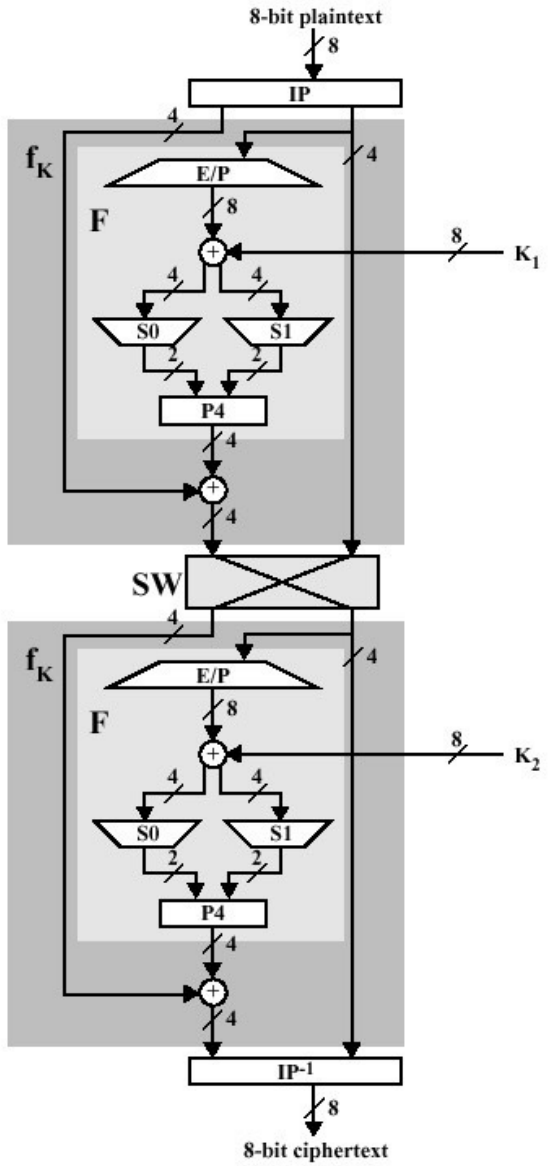
6) 5의 결과를 P-box(P4)에 입력하여 치환

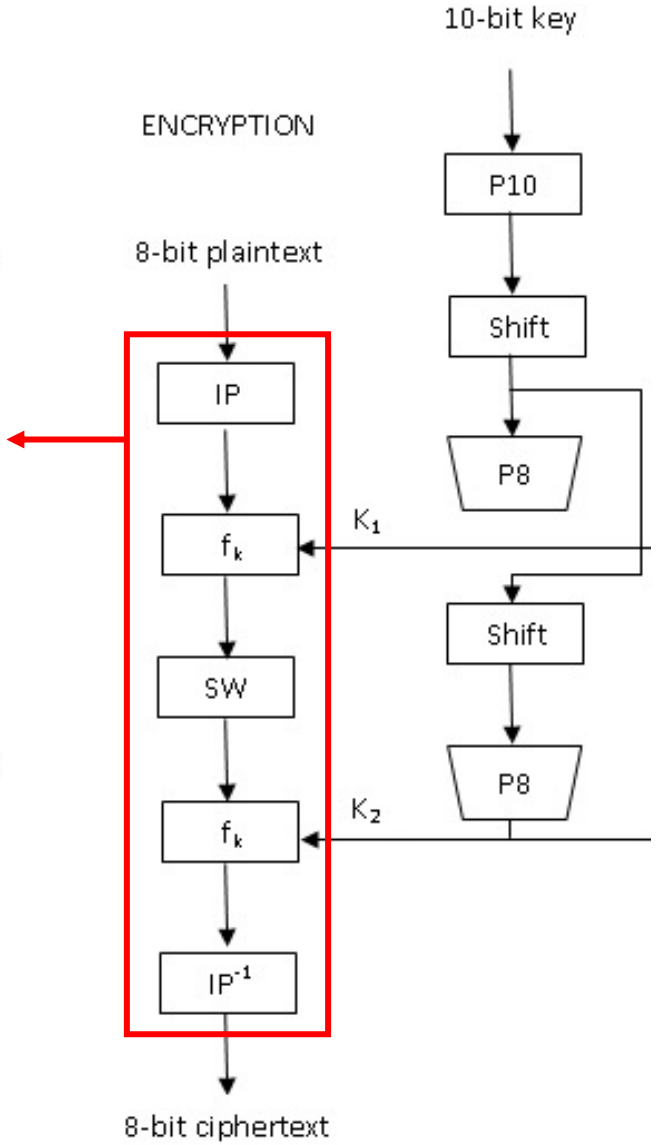
7) 6의 결과를 2의 L과 XOR

8) 7의 결과와 2의 R → SW(스위치함수)에 입력

9) 8의 결과를 가지고 2~8 과정과 동일하게 수행 (K1 → K2)

10) 9의 결과 → IP^{-1} 입력 ➔ 최종 암호문 생성

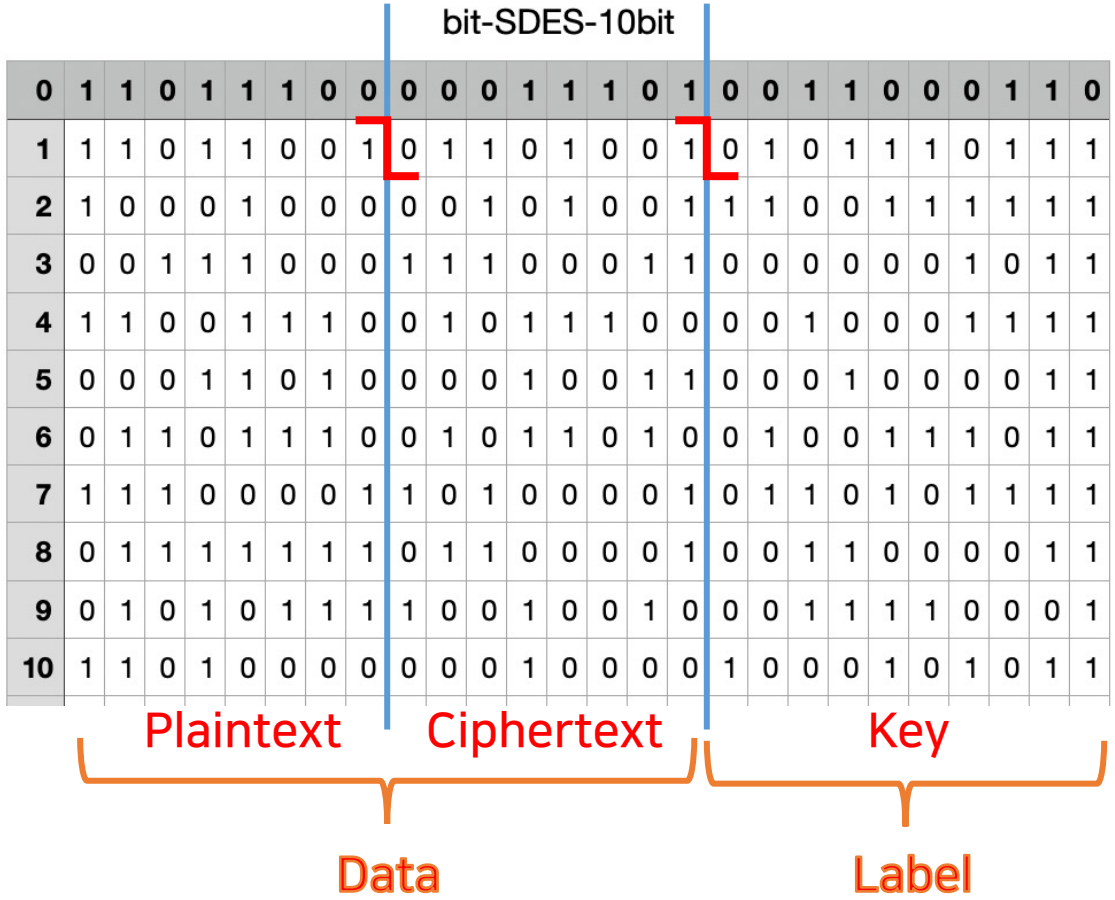




02. 데이터

- SDES python code 찾기 → 테스트 벡터 확인

✓ 100,000개의 (PT||CT||Key) CSV 파일 생성



0	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**
1	1	1	0	1	1	0	0	1	0	1	1	0	1	0	0	1	0	1	0	1	1	1	0	1	1	1
2	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1
3	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1	0	1	1
4	1	1	0	0	1	1	1	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	1
5	0	0	0	1	1	0	1	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0	0	1	1
6	0	1	1	0	1	1	1	0	0	1	0	1	1	0	1	0	0	1	0	0	1	1	1	0	1	1
7	1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	1	0	1	0	1	1	1	1
8	0	1	1	1	1	1	1	1	0	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1
9	0	1	0	1	0	1	1	1	1	0	0	1	0	0	1	0	0	0	1	1	1	1	0	0	0	1
10	1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	1	0	1	1

03. 모델 구조

```
x = tf.keras.layers.Conv1D(8, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(16, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(32, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(64, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(128, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(256, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(512, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(1024, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(2048, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)
```

CNN Layers

(conv1d는 1차원 Sequential data에 효과적)

Gated Linear Unit Layer

```
outputs = GatedLinearUnit(10)(x)

model = keras.Model(inputs=inputs, outputs=outputs)
opt = keras.optimizers.Adam(lr=0.0001)
model.compile(optimizer=opt, loss='mse', metrics=['mae'])
```

03. 모델 구조

```
Epoch 90/100
1782/1782 [==============================] - 9s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2252 - val_mae: 0.4499
Epoch 91/100
1782/1782 [==============================] - 9s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2250 - val_mae: 0.4499
Epoch 92/100
1782/1782 [==============================] - 8s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2250 - val_mae: 0.4499
Epoch 93/100
1782/1782 [==============================] - 8s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2250 - val_mae: 0.4499
Epoch 94/100
1782/1782 [==============================] - 9s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2251 - val_mae: 0.4499
Epoch 95/100
1782/1782 [==============================] - 8s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2250 - val_mae: 0.4499
Epoch 96/100
1782/1782 [==============================] - 8s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2250 - val_mae: 0.4499
Epoch 97/100
1782/1782 [==============================] - 8s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2250 - val_mae: 0.4499
Epoch 98/100
1782/1782 [==============================] - 8s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2251 - val_mae: 0.4499
Epoch 99/100
1782/1782 [==============================] - 8s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2251 - val_mae: 0.4499
Epoch 100/100
1782/1782 [==============================] - 9s 5ms/step - loss: 0.2251 - mae: 0.4499 - val_loss: 0.2250 - val_mae: 0.4499
```

```
[1.0, 0.588, 0.5093, 0.5163, 0.501, 0.8073, 0.5317, 0.514, 0.8423, 0.7263] / 0.6536
```

04. 하이퍼파라미터 튜닝

```
x = tf.keras.layers.Conv1D(8, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(16, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(32, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(64, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(128, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(256, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(512, kernel_size=9, strides=1, padding='same')(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(1024, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)

x = tf.keras.layers.Conv1D(2048, kernel_size=9, strides=1, padding='same')(inputs)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.ReLU()(x)
```

Layer의 개수와 채널 조절

kernel_size 조절

```
outputs = GatedLinearUnit(10)(x)

model = keras.Model(inputs=inputs, outputs=outputs)
opt = keras.optimizers.Adam(lr=0.0001)
model.compile(optimizer=opt, loss='mse', metrics=['mae'])
```

학습률 조절

04. 하이퍼파라미터 튜닝

<6-bit key size || cnn layers = 3>

Kernel_size	1	7	9
비트별 정확도	[**1.0, 1.0, 1.0, 1.0**, 0.4917, 0.5083, 0.5027, 0.5, 0.5163, 0.507]	[1.0, 1.0, 1.0, 1.0, 0.4883, 0.5027, 0.5133, 0.5343, 0.5633, 0.538]	[1.0, 1.0, 1.0, 1.0, 0.505, 0.5, 0.5063, 0.5027, 0.5377, 0.5167]
정확도	0.7026	0.714	0.7068
파라미터 수	4,148	8,660	24,628
CNN Layers	**3 Layers (16, 32, 64)**		
lr (학습률)	0.0001		

04. 하이퍼파라미터 튜닝

<6-bit key size || cnn layers = 5>

Kernel_size	1	3	7	9
비트별 정확도	[1.0, 1.0, 1.0, 1.0, 0.516, 0.4923, 0.5077, 0.508, 0.5237, 0.508]	[1.0, 1.0, 1.0, 1.0, 0.507, 0.52, 0.501, 0.5003, 0.513, 0.5033]	[1.0, 1.0, 1.0, 1.0, 0.487, 0.52, 0.501, 0.489, 0.5097, 0.5133]	[1.0, 1.0, 1.0, 1.0, 0.5007, 0.5177, 0.496, 0.505, 0.4953, 0.5177]
정확도	0.7056	0.7045	0.702	0.7032
파라미터 수	14,004	35,508	78,516	100,020
CNN Layers	**5 Layers (8, 16, 32, 64, 128)**			
lr (학습률)	0.0001			

04. 하이퍼파라미터 튜닝

<6-bit key size || cnn layers = 7>

Kernel_size	7
비트별 정확도	[1.0, 1.0, 1.0, 1.0, 0.5863, 0.587, 0.5067, 0.57, 0.506, 0.4967]
정확도	0.7253
파라미터 수	138,260
CNN Layers	**7 Layers** **(16, 32, 64, 128, 256, 512, 1024)**
lr (학습률)	0.0001

04. 하이퍼파라미터 튜닝

<6-bit key size || cnn layers = 7>

Kernel_size	1	3	7	9
비트별 정확도	[1.0, 1.0, 1.0, 1.0, 0.5083, 0.5647, 0.5993, 0.5917, 0.5947, 0.526]	[1.0, 1.0, 1.0, 1.0, 0.5773, 0.592, 0.515, 0.507, 0.545, 0.6427]	[1.0, 1.0, 1.0, 1.0, 0.564, 0.5477, 0.5883, 0.527, 0.5077, 0.5283]	[1.0, 1.0, 1.0, 1.0, 0.561, 0.5957, 0.6123, 0.5067, 0.54, 0.6717]
정확도	0.7385	0.7379	0.7263	0.7487
파라미터 수	79,892	145,428	276,500	342,036
CNN Layers	**7 Layers (32, 64, 128, 256, 512, 1024, 2048)**			
lr (학습률)	0.0001			

04. 하이퍼파라미터 튜닝

<6-bit key size || cnn layers = 9>

Kernel_size	1	3	7	9
비트별 정확도	[1.0, 1.0, 1.0, 1.0, 0.5087, 0.5803, 0.5587, 0.5013, 0.5973, 0.601]	[1.0, 1.0, 1.0, 1.0, 0.5217, 0.5053, 0.5257, 0.548, 0.5007, 0.5433]	[1.0, 1.0, 1.0, 1.0, 0.5573, 0.512, 0.606, 0.5837, 0.5253, 0.6723]	[1.0, 1.0, 1.0, 1.0, 0.5583, 0.5523, 0.6063, 0.5037, 0.6643, 0.6263]
정확도	0.7347	0.7145	0.7457	0.7511
파라미터 수	79,892	145,428	276,500	342,036
CNN Layers	**9 Layers (8, 16, 32, 64, 128, 256, 512, 1024, 2048)**			
lr (학습률)	0.0001			

➔ 9 Layers && kernel_size가 클수록 정확도가 높아짐

04. 하이퍼파라미터 튜닝

key	6-bit	7-bit	8-bit	9-bit
비트별 정확도	[1.0, 1.0, 1.0, 1.0, 0.5583, 0.5523, 0.6063, 0.5037, 0.6643, 0.6263]	[1.0, 1.0, 1.0, 0.5, 0.5513, 0.7343, 0.4817, 0.6187, 0.642, 0.8047]	[1.0, 1.0, 0.573, 0.5, 0.506, 0.7943, 0.513, 0.5, 0.651, 0.6607]	[1.0, 0.588, 0.5093, 0.5163, 0.501, 0.8073, 0.5317, 0.514, 0.8423, 0.7263]
정확도	0.7511	0.7333	0.6698	0.6536
epoch	100			
Kernel_size	9			
CNN Layers	9 Layers (8, 16, 32, 64, 128, 256, 512, 1024, 2048)			

감사합니다