Applied Information Security and Cryptography

Lab 02: A simplified version of AES

In this lab, you will experiment with a simplified version of AES proposed in “Musa, Mohammad & Schaefer, Edward & Wedig, Stephen. (2003). A simplified AES algorithm and its linear and differential cryptanalyses. CRYPTOLOGIA. 27. 148-177.” Simplified AES uses a block of 16 bits and a key of 16 bits. Python implementation of simplified AES has been derived from this one: https://jhafranco.com/2012/02/11/simplified-aes-implementation-in-python/ . You will evaluate avalanche effect when simplified AES is implemented using different number of rounds. Then, you will see how improperly implemented versions of simplified AES can be easily broken.

Simplified AES

The Python script AISC_02.py provides an implementation of simplified AES. The original version of simplified AES has only two rounds: after the initial AddKey, there is a generic round composed by the usual four operations (SBox, ShiftRows, MixColumns, AddKey) and a final round composed by three operations (SBox, ShiftRows, AddKey). You can take a look at the code to see how different functions are implemented. The rationale is very similar to original AES. The state of simplified AES is a 2x2 matrix of 4-bit values (nibbles), S-Box are defined for 4-bit values, ShiftRows, MixColumns, and AddKey perform similar operations as original AES. The function mult(p1,p2) is a nice example of how to compute multiplications in GF(2^n).

To use simplified AES, first define a key as a 16-bit integer, for example:

```
key = 0b0100101011110101
```

then expand the key to generate the different subkeys for each round:

```
keyExp(key)
```

The subkeys are generated in a global array w[] containing 10 bytes: w[0] and w[1] contain the subkey for the first AddKey, w[2] and w[3] contain the subkey for the first round, w[4] and w[5] contain the subkey for the second round. The other pairs of w[] values can be used as keys for additional rounds.

To encrypt a plaintext, expressed as a 16-bit integer, use encrypt:

```
plaintext = 0b1101011100101000
ciphertext = encrypt(plaintext)
```

To decrypt a ciphertext, use decrypt:

```
plaintext2 = decrypt(ciphertext)
assert plaintext2 == plaintext
```

Encryption and decryption will use the last key expanded by keyExp(key).

Avalanche effect

Write a script to test the avalanche effect in simplified AES. The script should perform two different experiments.

In the first experiment, choose a key, then generate a random plaintext and encrypt it. Now, flip a bit of the plaintext in a random position and generate a second encryption. Compare the two encryptions and record the number of bits in which they differ (this is the Hamming distance of the two ciphertexts). Repeat the above experiment 1000 times and record the average Hamming distance between ciphertexts.

In the second experiment, choose a plaintext, then generate a random key and expand it. Encrypt the plaintext under that key. Now, flip a bit of the key in a random position and expand the new key. Encrypt the same plaintext under the new key. Compute the Hamming distance of the two encryptions. Repeat the above experiment 1000 times and record the average Hamming distance between ciphertexts.

Generating a random n-bit value can be done using the function getrandbits() in the library random:

```
import random

plaintext = random.getrandbits(16)
```

Flipping a random bit can be achieved by generating an error vector with just one bit set to one (e.g., shift left 1 by a random value) and computing the XOR with the error vector:

```
error = 1 << random.randrange(16)
plaintext2 = plaintext ^ error
```

The file AISC_02.py provides some sample code. To save time, a function computing the Hamming distance of two values is also provided, for example:

```
assert hamming(plaintext, plaintext2) == 1
```

Now, you should have an idea of how many ciphertext bits change on average when you change one bit in the plaintext or you change one bit in the key. See how the number of rounds affects this behaviour. Write a version of simplified AES using three rounds instead of two. You should include an additional generic round with four operations, where this round consumes an additional subkey from the expanded key. So, three-round simplified AES should use subkeys from (w[0],w[1]) to (w[6],w[7]). Verify that the implementation is correct by performing encryption and decryption. Repeat the above experiments with three-round simplified AES and record the avalanche effect in this case.

Write also a four-round version of simplified AES, using all subkeys, and run the experiments with this version too.

For the last two experiments, assume that a lazy programmer didn't have time to implement all of AES functions. Rewrite the four-round simplified AES without using ShiftRow and MixColumns. In this case, a round will be simply the S-Box substitution followed by AddKey. We will name this version *lazy simplified AES*. Do the avalanche effect experiments on this lazy version.

Now, assume that the lazy programmer didn't even implement the key schedule. Use the same key, i.e., w[0], w[1], in all rounds. We will name this version *very lazy simplified AES*. Repeat the avalanche effect experiment on this very lazy version. These two last experiments should give you some hints on why we need a substitution and permutation network and a key schedule.

Improperly implemented block cipher

With the aim of obtaining a very fast encryption algorithm, a programmer without much expertise in cryptography devised an extremely simplified AES version implemented by the functions encrypt_foo() and decrypt_foo() in AISC_02.py. As you can see, AddKey is performed only at the end. A text file has been encrypted with this scheme and saved in ciphertext.txt.

You obtained the following plaintext-ciphertext pair encrypted using the same key:

```
known_plain =  0b0111001001101110
known_cipher = 0b0010111000001101
```

Decrypt ciphertext.txt. (Brute force is not allowed)

Some notes on the format. For the encryption, two consecutive characters c0,c1 of the plaintext string have been mapped to bytes and converted to a 16-bit integer as follows:

```
plaintext = (ord(c0) << 8) + ord(c1)
```

The corresponding 16-bit ciphertext has been converted to a bytearray as follows:

```
encryption = bytearray()

encryption.append((ciphertext & 0xff00) >> 8)
encryption.append(ciphertext & 0x00ff)
```

Finally, the bytearray has been written to ciphertext.txt using base64 encoding:

```
with open("ciphertext.txt", "w") as text_file:
    text_file.write(base64.b64encode(encryption).decode("utf-8"))
```

When opening ciphertext.txt, you can recover the same bytearray as:

```
with open("ciphertext.txt", "r") as text_file:
        encryption = base64.b64decode(text_file.read())
```

Questions (Answer these in your report)

Report the results of the avalanche effect for the different tested versions (simplified AES, three-round simplified AES, four-round simplified AES, lazy simplified AES, very lazy simplified AES) and the different experiments (average number of ciphertext bits that differ after changing a single bit in the plaintext, average number of ciphertext bits that differ after changing a single bit in the key).

Looking at the results, which is the minimum number of rounds for having a sufficient avalanche effect?

Why the lazy version, even though it uses four rounds, has a limited avalanche effect when the plaintext changes? You can inspect some of the ciphertexts before and after changing the plaintext to gain some insight.

When the key changes, only the very lazy version seems to have a limited avalanche effect. Can you find a reason for this behaviour? You can inspect the subkeys in the key schedule to gain some insight.

Describe how you decrypted ciphertext.txt. Which was the main weakness of the “foo encryption”?

Bonus question: under KPA, ciphertext.txt is easily broken. How would you decrypt it without having a known plaintext? Is there a more efficient way than brute force?