Notebook

October 21, 2024

1 Problem 2. AntCipher 2.0

1.1 Problem:

- Giving this CNF C:

$C = (x_1 \vee x_2 \vee \neg x_5) \wedge (\neg x_1 \vee \neg x_2 \vee x_5) \wedge (x_1 \vee x_3 \vee \neg x_5) \wedge (\neg x_1 \vee \neg x_3 \vee x_5) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_3 \vee x_5) \wedge (x_1 \vee x_2 \vee x_5) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_5) \wedge (x_1 \vee x_3 \vee x_5) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_5) \wedge (x_2 \vee x_3 \vee x_5) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_5) \wedge (x_1 \vee x_4 \vee x_6) \wedge (\neg x_1 \vee \neg x_4 \vee x_6) \wedge (x_2 \vee x_4 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_4 \vee x_6) \wedge (x_3 \vee x_4 \vee x_6) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_6) \wedge (x_1 \vee x_2 \vee \neg x_6) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_6) \wedge (x_2 \vee x_3 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_6) \wedge (x_1 \vee \neg x_4 \vee x_7) \wedge (\neg x_1 \vee \neg x_4 \vee \neg x_7) \wedge (x_2 \vee x_4 \vee \neg x_7) \wedge (\neg x_2 \vee \neg x_4 \vee x_7) \wedge (x_3 \vee x_4 \vee \neg x_8) \wedge (\neg x_3 \vee \neg x_4 \vee x_8) \wedge (x_2 \vee x_4 \vee \neg x_8) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_8)$

- This CNF represents a nonlinear function F_C where `C = 1`, taking a 4-bit input `(x1, x2, x3, x4)` and producing a 4-bit output `(x5, x6, x7, x8)`. In the i-th iteration of the cipher, a 64-bit value of R is divided into 16 4-bit sequences, which are given to F_C as inputs. Then 16 4-bit outputs are produced and concatenated thus forming a 64-bit K_i that is written to R and is used as a keystream. Given some infomation:
- 1704th ciphertext: `1001 1000 0011 1101 0110 0011 1101 0101 1011 0011 1011 0111 0000 0000 1000 0011`
- The 1702nd keystream: `0101 1001 1111 0011 00X1 X111 1X00 00X0 111X X000 XXXX XXXX XXXX XXXX XXXX XXXX`
- The 1703rd keystream: `XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX X111 000X X010 01X1 0X10 0101 0000 1111`

Our goal is to find the plaintext in iteration 1704 to locate the ant.

1.2 Solution:

We can simulate the nonlinear function F_C by this python function:

[6]:
```python
def boolean_function(x1, x2, x3, x4, x5, x6, x7, x8):
    clause1 = (x1 or x2 or (not x5))
    clause2 = ((not x1) or (not x2) or x5)
    clause3 = (x1 or x3 or (not x5))
    clause4 = ((not x1) or (not x3) or x5)
    clause5 = (x2 or x3 or (not x5))
    clause6 = ((not x2) or (not x3) or x5)
    clause7 = (x1 or x2 or (not x6))
    clause8 = ((not x1) or (not x2) or x6)
```

```
    clause9 = (x1 or x4 or (not x6))
    clause10 = ((not x1) or (not x4) or x6)
    clause11 = (x2 or x4 or (not x6))
    clause12 = ((not x2) or (not x4) or x6)
    clause13 = (x1 or x3 or (not x7))
    clause14 = ((not x1) or (not x3) or x7)
    clause15 = (x1 or x4 or (not x7))
    clause16 = ((not x1) or (not x4) or x7)
    clause17 = (x3 or x4 or (not x7))
    clause18 = ((not x3 )or (not x4) or x7)
    clause19 = (x2 or x3 or (not x8))
    clause20 = ((not x2) or (not x3) or x8)
    clause21 = (x2 or x4 or (not x8))
    clause22 = ((not x2 )or (not x4) or x8)
    clause23 = (x3 or x4 or (not x8))
    clause24 = ((not x3 )or (not x4) or x8)
    return (clause1 and clause2 and clause3 and clause4 and clause5 and clause6␣
↪and
            clause7 and clause8 and clause9 and clause10 and clause11 and␣
↪clause12 and
            clause13 and clause14 and clause15 and clause16 and clause17 and␣
↪clause18 and
            clause19 and clause20 and clause21 and clause22 and clause23 and␣
↪clause24)
```

With this function, we can form a mapping table which 4-bit input and 4-bit output satifying `C = 1`

[30]:
```
mapping = dict()
import itertools
for x1, x2, x3, x4, x5, x6, x7, x8 in itertools.product([False, True],␣
↪repeat=8):
    result = boolean_function(x1, x2, x3, x4, x5, x6, x7, x8)
    if result == 1:
        l1 = "".join(map(str,(map(int, [x1, x2, x3, x4]))))
        l2 = "".join(map(str,(map(int, [x5, x6, x7, x8]))))
        mapping[l1] = l2
```

[31]:
```
mapping
```

[31]:
```
{'0000': '0000',
 '0001': '0000',
 '0010': '0000',
 '0011': '0011',
 '0100': '0000',
 '0101': '0101',
 '0110': '1001',
```

```
 '0111': '1111',
 '1000': '0000',
 '1001': '0110',
 '1010': '1010',
 '1011': '1111',
 '1100': '1100',
 '1101': '1111',
 '1110': '1111',
 '1111': '1111'}
```

- Suppose that we are generating the n-th keystream to the (n+1)-th keystream using this mapping. We can see that all block in n-th keystream must be one of the output of the mapping because it 's the keystream generated by (n-1)-th keystream. So that in the mapping, some keys are not necessary.
- We can reduce the mapping like this.

[32]:
```
values = list(mapping.values())
keys = list(mapping.keys())
for k in keys:
    if k not in values:
        mapping.pop(k)
print(mapping)
```

```
{'0000': '0000', '0011': '0011', '0101': '0101', '0110': '1001', '1001': '0110',
'1010': '1010', '1100': '1100', '1111': '1111'}
```

[33]:
```
def recover(K1, K2):
    if all(c == "X" for c in K2):
        if 'X' in K1:
            # case when K2 is full of X
            # replace the unknown X in K2 with 0 and 1 to check possible␣
 ↪mappings
            K1_0 = K1.replace('X', '0')
            K1_1 = K1.replace('X', '1')
            if K1_0 in mapping.keys():
                # check if K1_0 is in mapping.keys()
                return mapping[K1_0]
            else:
                # if not, then K1_1 must be in mapping.keys()
                return mapping[K1_1]
        else:
            return mapping[K1]
    elif 'X' not in K2:
        # case when K2 is cleared
        return K2
    else:
        # case when K2 has 1 character X
        # replace X in K2 with 0 and 1
```

```python
        K2_0 = K2.replace('X', '0')
        K2_1 = K2.replace('X', '1')
        if K2_0 in mapping.keys():
            # check if K2_0 is in mapping.keys()
            return K2_0
        else:
            # if not, then K2_1 must be in mapping.keys()
            return K2_1
```

Now, let's divide the keystream into 4-bit blocks and recover all values

[34]:
```python
K_1702 = ("0101 1001 1111 0011 00X1 X111 1X00 00X0 111X X000 XXXX XXXX XXXX␣
 ↪XXXX XXXX XXXX").split(" ")
K_1703 = ("XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX X111 000X X010 01X1 0X10␣
 ↪0101 0000 1111").split(" ")
recovered = ''
ciphertext = 0b1001100000111101011000111101010110110011101101110000000010000011
for i in range(len(K_1702)):
    recovered += recover(K_1702[i], K_1703[i])
print(recovered)
```

```
0101011011110011001111111100000011110000101001010110010100001111
```

With the recovered `K_1703`, we can easily generate the 1704th keystream, XOR it with the given ciphertext to recover the plaintext, and convert it into a pair of floating-point values.

[35]:
```python
K_1704 = ""
for i in range(0, len(recovered), 4):
    K_1704 += mapping[recovered[i:i+4]]
print(K_1704)
```

```
0101100111110011001111111100000011110000101001011001010100001111
```

[36]:
```python
plaintext = ciphertext ^ int(K_1704, 2)
import struct
latitude = plaintext >> 32
longtitude = plaintext % (1<<32)

latitude = struct.unpack('!f', struct.pack('!I', latitude))[0]
longtitude = struct.unpack('!f', struct.pack('!I', longtitude))[0]
print(latitude, longtitude)
```

```
-25.79496192932129 146.58416748046875
```

So, the latitude and longitude are -25.79496192932129 and 146.58416748046875 respectively.