NSUCRYPTO2024

Problem 7: A nonlinear generator

October 21, 2024

Solution

$A(1) = 01111100001110000110010001011000110000110011110$

$B(1) = 11001100110100010101010001011101000110110100101 10$

We approach this problem by recursively deducing the earlier states of the registers, starting from the known state at time $t = 8192$. We try all possible combinations for the first bits of $A(t)$ and $B(t)$ at each time step and verify whether they satisfy the register's feedback functions. If a valid combination is found, we proceed to deduce the states at time $t - 1$, continuing until we reach $t = 1$.

The feedback functions are checked using the following function:

$$\begin{aligned}\text{check}(t, sA, sB) = ((sA[0]\&sA[1]) \oplus sA[12] \oplus sA[43] = sA[47]) \\ \wedge ((sB[0]\&sB[1]) \oplus sB[10] \oplus sB[47] = sB[49]) \\ \wedge ((sA[8]\&sA[9]) \oplus sA[4] \oplus sA[1] \oplus (sB[5]\&sB[6]) \oplus sB[8] \oplus sB[1] = K[t])\end{aligned}$$

This function checks whether the current states of registers A and B satisfy the generator's feedback rules and whether they produce the correct keystream bit $K[t]$ at time t.

Below is the Python script used to implement this approach:

```python
def check(t, sA, sB):
    res = (((sA[0] & sA[1]) ^ sA[12] ^ sA[43]) == sA[47])
    res = res and (((sB[0] & sB[1]) ^ sB[10] ^ sB[47]) == sB[49])
    res = res and (((sA[8] & sA[9]) ^ sA[4] ^ sA[1] ^ (sB[5] & sB[6]) ^ sB[8] ^ sB[1]) == K[t])
    return res

ans = []

def solve(t, sA, sB):
    if t == 1:
        ans.append((sA[1:], sB[1:]))
        return

    for i, j in product(range(2), range(2)):
        sA[0] = i
        sB[0] = j

        if check(t, sA, sB):
            solve(t - 1, [None] + sA[:47], [None] + sB[:49])


solve(8192, [None] + A, [None] + B)
state_A, state_B = ans[0]
```

Figure 1: Finding initial states

The function `solve` recursively computes the earlier states, starting from $t = 8192$ and trying all possible combinations for $sA[0]$ and $sB[0]$ to satisfy the feedback rules.

After recovering the initial states $A(1)$ and $B(1)$, we use them to simulate the generator and generate the keystream. We then compare the generated keystream with the one provided in the file to ensure that the recovered states are correct.

We define a class called `Generator` that simulates the behavior of the two shift registers, $A(t)$ and $B(t)$. The shift registers are updated based on the feedback rules described earlier, and at each time step, they produce one bit of the keystream.

The Python code for the `Generator` class is as follows:

```python
class Generator:
    def __init__(self, state_A, state_B):
        self.state_A = state_A
        self.state_B = state_B

    def _clock_A(self):
        b = self.state_A[0] ^ (self.state_A[7] & self.state_A[8]) ^ self.state_A[3]
        self.state_A = self.state_A[1:] + [(self.state_A[0] & self.state_A[1]) ^ self.state_A[12] ^ self.state_A[43]]
        return b

    def _clock_B(self):
        b = self.state_B[0] ^ (self.state_B[4] & self.state_B[5]) ^ self.state_B[7]
        self.state_B = self.state_B[1:] + [(self.state_B[0] & self.state_B[1]) ^ self.state_B[10] ^ self.state_B[47]]
        return b

    def _clock(self):
        return self._clock_A() ^ self._clock_B()
```

Figure 2: Diagram of the Generator class implementation.

Once the Generator is initialized with the recovered initial states, we generate the keystream by clocking the Generator 8192 times (the length of the known keystream). The generated keystream is then compared to the provided keystream in `keystream.txt`.

The Python code to generate the keystream and compare it with the known keystream is as follows:

```python
key_stream = open('./keystream.txt', 'r').read().strip()
g = Generator(state_A, state_B)
k = ''

for _ in range(8192):
    k += str(g._clock())

print(key_stream == k) # True
```

Figure 3: Keystream comparison process.

This code reads the known keystream from the file, initializes the Generator with the recovered states $A(1)$ and $B(1)$, and then generates the keystream bit by bit. Finally, it checks whether the generated keystream matches the given keystream. If the two match, then the initial states have been correctly recovered.

Please refer to the solution script for more details on NSUCRYPTO2024 Problem 7.