CS4236 Project 1 Report

Keys found: 41 (+3) = 44

Machine configuration: Laptop, CPU: i7-3630QM @2.40 GHz, Memory: 8GB DDR3

# Background

As described in the Project Description, the aim of this project is to have a total break on WEP encryption. In WEP, encrypted frames are sent. However, the IV used is stored in clear in the frame. Thus, from the attacker’s point of view, the adversaries can see the IVs and output bytes. Using this knowledge, we want to break the encryption key.

There will be 60 data sets using different number of keys (8, 10, 12, 14, 16 and 18) and different number of known data. Each data is a tuple with 4 values, (v1, v2, v3, x) where v1, v2 and v3 are the IV and x is the first output byte.

# Literature Review



**Fig. 1. The Key Scheduling Algorithm and Pseudo Random Generation Algorithm**

WEP is using RC4 in their encryption scheme. In RC4 there are two notable algorithms, which are the Key Scheduling Algorithm (KSA) and Pseudo Random Generation Algorithm (PGRA).

As mention in the Project Description and Project Background during lecture, we know that we can exploit KSA to get the keys.

During lecture, we know how Fluhrer, Mantin, and Shamir (FMS) exploit the KSA to break the RC4 key. In this project, we will use Korek’s method. Korek described in his paper 17 attack methods in great details including the generalization of FMS attack (Korek attack 1).

The first 8 methods require the knowledge of the first output byte from the PGRA. While the next 9 methods require the knowledge of the second output byte from the PGRA. Thus, in this project, I will only use the first 8 methods described by Korek to break the RC4 encryption key since we only know the IV and first output byte from the data tuples.

Below is a short description of the conditions or restrictions for the Korek attack methods to be successful.

Si[a] is the position of element a in the permutation table S

x is the first output byte generated from PGRA

p is the current key position to be found

|  |  |  |
| --- | --- | --- |
| Attack No. | Probability | Condition |
| Korek 1 | 5.07% | S[1] < p && mod(S[1] + S[S[1]], N) == p && Si[X] != 1 && Si[X] != 4 &&  Si[X] != S[S[1]] |
| Korek 2 | 13.75% | S[1] == p && x == p |
| Korek 3 | 13.75% | S[1] == p && x == mod(1-p, N) |
| Korek 4 | 5.07% | S[1] == p && x != mod((1-p), N) && x != p && Si[X] < p &&  Si[mod(Si[X] - p, N)] != 1 |
| Korek 5 | 5.07% | Si[X] == 2 && S[p] == 1 |
| Korek 6 | 13.75% | S[p] == p && S[1] == 0 && x == p |
| Korek 7 | 13.75% | S[p] == p && x == S[1] && S[1] == mod(1-p, N) |
| Korek 8 | 5.07% | S[p] == p && S[1] >= (N - p) % N && S[1] == mod(Si[X]- p, N) && Si[X]!= 1 |

# Implementation

Code is written in C++ and run using PowerShell script. Notable algorithm for my best performing (non-manual) performance:

To crack the key

findKey(int B){

p = B+3;

if(p >= L – 3){

brute force for the last 3 bytes using key [0,89]

}

for all data tuples:

if match condition 1: add mod(Si[x] - S[p] - j, N) as candidate

if match condition 2: add mod(Si[0] - S[p] - j, N) as candidate

if match condition 3: add mod(Si[x] - S[p] - j, N) as candidate

if match condition 4: add mod(Si[mod(Si[x] - p, N)] - S[p] - j, N) as candidate

if match condition 5: add mod(1 - S[p] - j, N) as candidate

if match condition 6: add mod(1 - S[p] - j, N) as candidate

if match condition 7: add mod(1 - S[p] - j, N) as candidate

if match condition 8: add mod(1 - S[p] - j, N) as candidate

K[p] = candidate that occurs the most

findKey(B+1);

}

After a key is found, there’s also a key checking algorithm

confirmKey() {

for each different IV appended with the key

do KSA using the full key

if the known first output != first output of PGRA

return false

return true

}

I ran the code under 3 different environments:

1. Using only FMS (found 9 keys) : Run time ~10 min  
   - Output file: result1.txt
2. Using 1 of 8 conditions using if-else (found 29 keys) : Run time ~12 min  
   - Output file: result2.txt
3. Using all 8 conditions using if (found 41 keys) : Run time ~12 min  
   - Output file: result3.txt

For keys those cannot be found automatically, I did a brute force method. After I found the candidates, I tried all candidates manually from the highest occurring candidates. Using this method, I managed to found 3 keys.

This is a kind of backtracking method but in this case, it is a manual backtracking. I did not implement an automated backtracking algorithm due to time constraint and I believe that it will not improve the performance by much.

# Data

|  |  |  |  |
| --- | --- | --- | --- |
| File | Key Size | No. of Tuples | Key |
| A00 | 8 | 5,000,000 | 63 63 88 73 12 |
| A01 | 8 | 3,000,000 | 16 80 1 89 86 |
| A02 | 8 | 2,000,000 | 30 56 14 22 53 |
| A03 | 8 | 1,500,000 | 87 18 61 71 1 |
| A04 | 8 | 1,000,000 | 30 3 76 6 37 |
| A05 | 8 | 750,000 | 17 59 72 55 38 |
| A06 | 8 | 500,000 | 12 12 11 47 8 |
| A07 | 8 | 300,000 | - |
| A08 | 8 | 200,000 | 89 11 10 54 19 |
| A09 | 8 | 100,000 | 65 15 53 17 39  (brute force) |
| A10 | 10 | 5,000,000 | 17 78 61 20 77 52 68 |
| A11 | 10 | 3,000,000 | 21 80 32 79 56 16 54 |
| A12 | 10 | 2,000,000 | 11 66 87 5 47 50 65 |
| A13 | 10 | 1,500,000 | 68 10 52 48 61 47 7 |
| A14 | 10 | 1,000,000 | 20 10 34 33 30 65 89 |
| A15 | 10 | 750,000 | - |
| A16 | 10 | 500,000 | 5 61 52 12 59 76 15 |
| A17 | 10 | 300,000 | 57 46 24 0 64 43 43 |
| A18 | 10 | 200,000 | 31 4 25 81 61 77 21 |
| A19 | 10 | 100,000 | - |
| A20 | 12 | 5,000,000 | 23 67 17 71 24 23 0 35 52 |
| A21 | 12 | 3,000,000 | 34 60 27 57 19 30 9 87 55 |
| A22 | 12 | 2,000,000 | 46 48 36 13 23 7 18 80 10 |
| A23 | 12 | 1,500,000 | 68 29 14 60 2 73 13 79 37 |
| A24 | 12 | 1,000,000 | 20 5 57 79 55 44 78 85 0 |
| A25 | 12 | 750,000 | 89 88 68 15 42 3 66 74 85 |
| A26 | 12 | 500,000 | 58 78 53 44 73 41 10 14 75 |
| A27 | 12 | 300,000 | 30 13 20 35 49 64 51 64 88 |
| A28 | 12 | 200,000 | 75 19 83 6 77 18 32 76 83 |
| A29 | 12 | 100,000 | - |
| A30 | 14 | 5,000,000 | 54 71 11 34 71 28 15 49 32 9 1 |
| A31 | 14 | 3,000,000 | - |
| A32 | 14 | 2,000,000 | 81 6 11 45 67 44 81 6 32 55 60 |
| A33 | 14 | 1,500,000 | 27 85 33 45 37 46 83 12 35 85 74 |
| A34 | 14 | 1,000,000 | 26 45 22 40 76 59 74 23 73 57 80 |
| A35 | 14 | 750,000 | - |
| A36 | 14 | 500,000 | 67 23 70 72 64 87 47 2 85 46 26 |
| A37 | 14 | 300,000 | - |
| A38 | 14 | 200,000 | - |
| A39 | 14 | 100,000 | - |
| A40 | 16 | 5,000,000 | 6 12 49 66 37 78 66 57 11 84 70 5 44 |
| A41 | 16 | 3,000,000 | 5 62 42 12 42 5 21 72 48 7 17 55 17 |
| A42 | 16 | 2,000,000 | 23 72 63 66 20 34 60 85 57 15 15 54 16  (brute force) |
| A43 | 16 | 1,500,000 | 71 71 71 28 21 38 68 63 54 40 83 23 65 |
| A44 | 16 | 1,000,000 | - |
| A45 | 16 | 750,000 | 34 5 47 23 6 37 23 58 32 11 82 49 51 |
| A46 | 16 | 500,000 | - |
| A47 | 16 | 300,000 | - |
| A48 | 16 | 200,000 | - |
| A49 | 16 | 100,000 | - |
| A50 | 18 | 5,000,000 | 1 12 53 56 79 61 2 52 49 43 64 23 26 43 27 |
| A51 | 18 | 3,000,000 | 67 83 0 24 67 3 63 81 47 66 86 30 43 6 17 |
| A52 | 18 | 2,000,000 | 19 8 52 6 32 83 54 87 33 86 40 88 42 70 46 |
| A53 | 18 | 1,500,000 | 4 72 39 88 37 32 19 2 24 35 65 15 52 57 84 |
| A54 | 18 | 1,000,000 | 41 25 37 72 54 28 4 41 27 74 10 39 27 81 30 |
| A55 | 18 | 750,000 | 14 78 26 53 23 25 26 39 21 17 44 34 33 18 78 |
| A56 | 18 | 500,000 | 43 56 85 17 56 88 5 13 14 10 50 48 71 69 42 |
| A57 | 18 | 300,000 | 38 35 59 31 74 43 29 74 72 79 6 62 76 59 62  (brute force) |
| A58 | 18 | 200,000 | - |
| A59 | 18 | 100,000 | - |

As we can see, generally, with lower known data (no. of tuples), it is harder to break the encryption and get the key. This is mainly due to not enough statistical data to derive the key.

With more data, there will be more weak IVs occurring and using these weak IVs, the probability of successful attacks will be higher.

For the keys those cannot be found, I suspect that very little weak IVs occurring throughout the data. Hence, it is very difficult for the adversaries to crack the key.

# Conclusion

As discussed earlier, the main weakness of RC4 is on the KSA. The adversaries just need to know a handful pairs of IV and output bytes and the key can be cracked. This is due to the predictability of the permutation of the array. Furthermore, with the given constraints, and given that I can find about 66% of the keys in under 15 minutes, WEP which use RC4 as their encryption scheme is not safe at all.