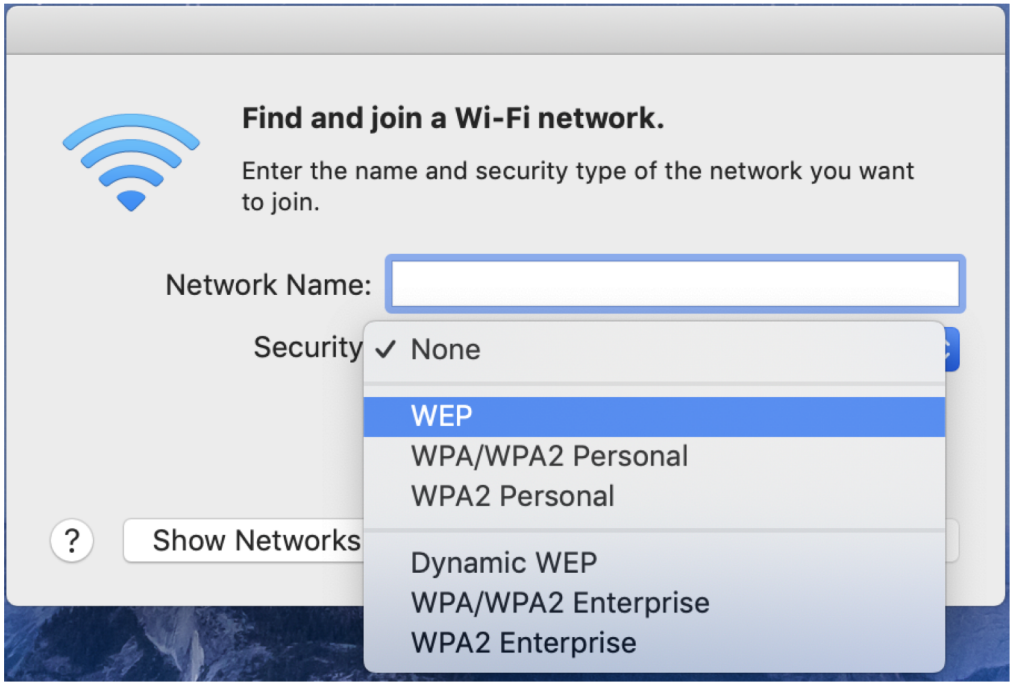
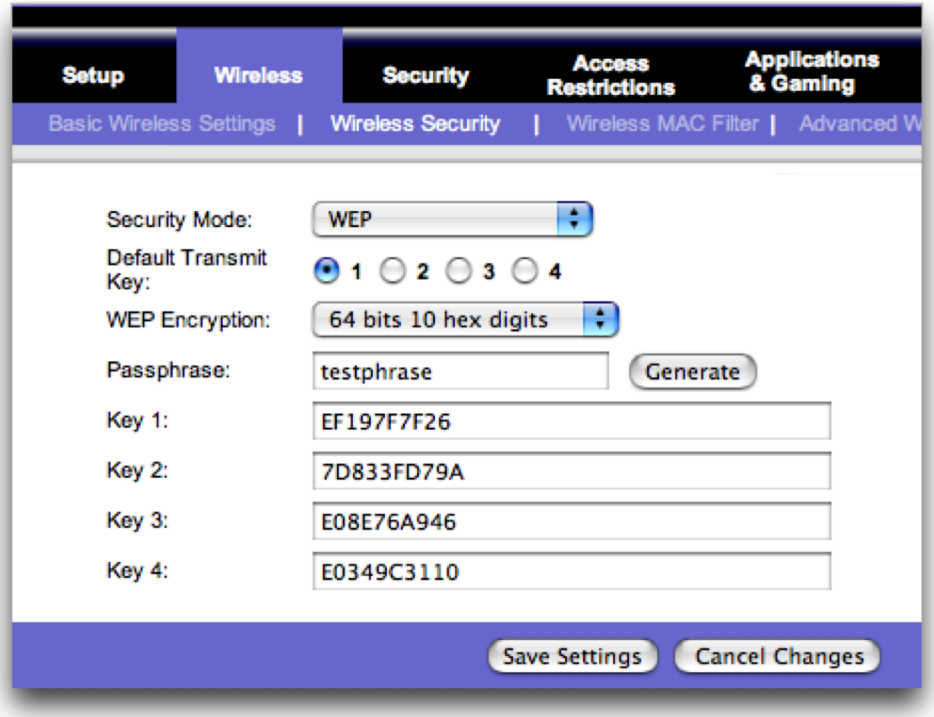
FMS Attack

Based on WEP, which uses RC4 for confidentiality.

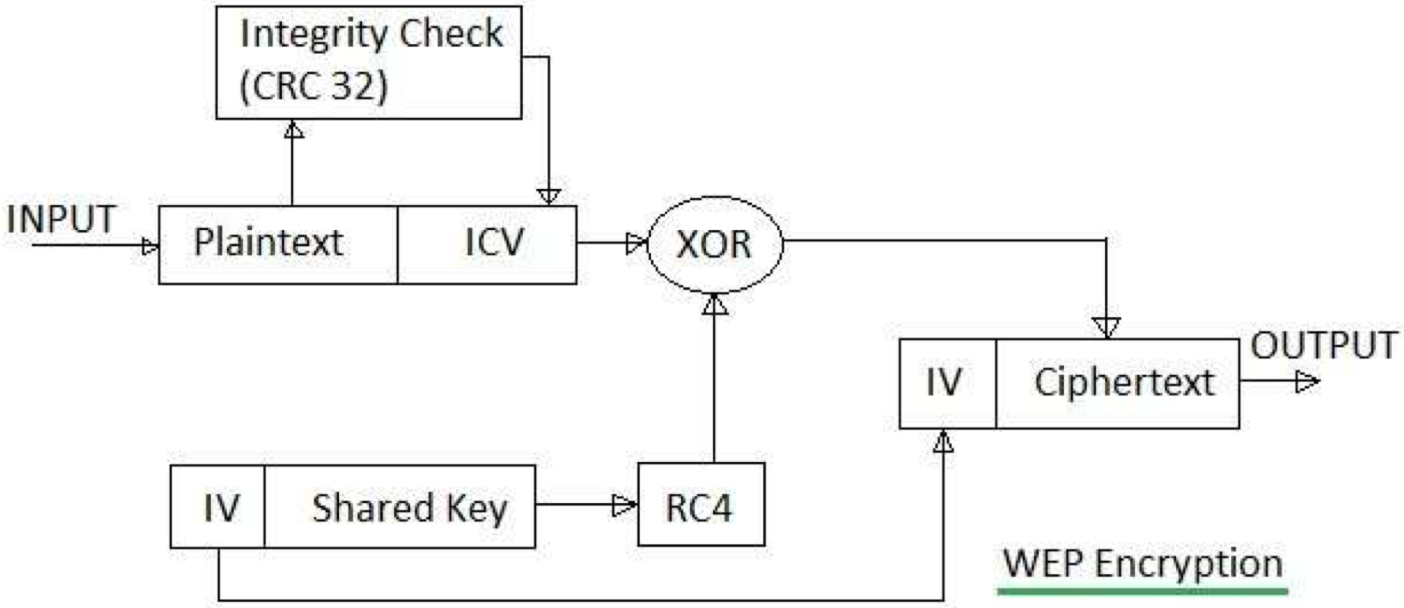




Picture: https://arstechnica.com/information-technology/2008/11/wpa-cracked/

WEP

- WEP was included as the privacy component of the original IEEE 802.11 standard ratified in 1997.
- It uses RC4 for confidentiality and CRC-32 checksum for integrity.

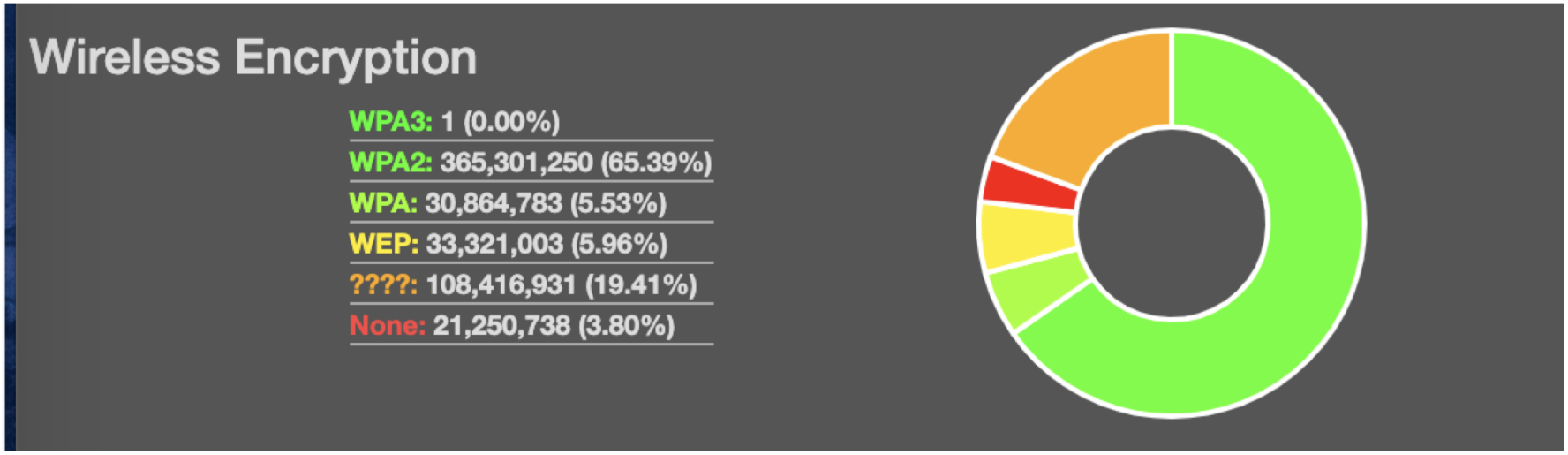


Picture: http://www.rfwireless-world.com/Terminology/WEP-vs-WPA-vs-WPA2.html

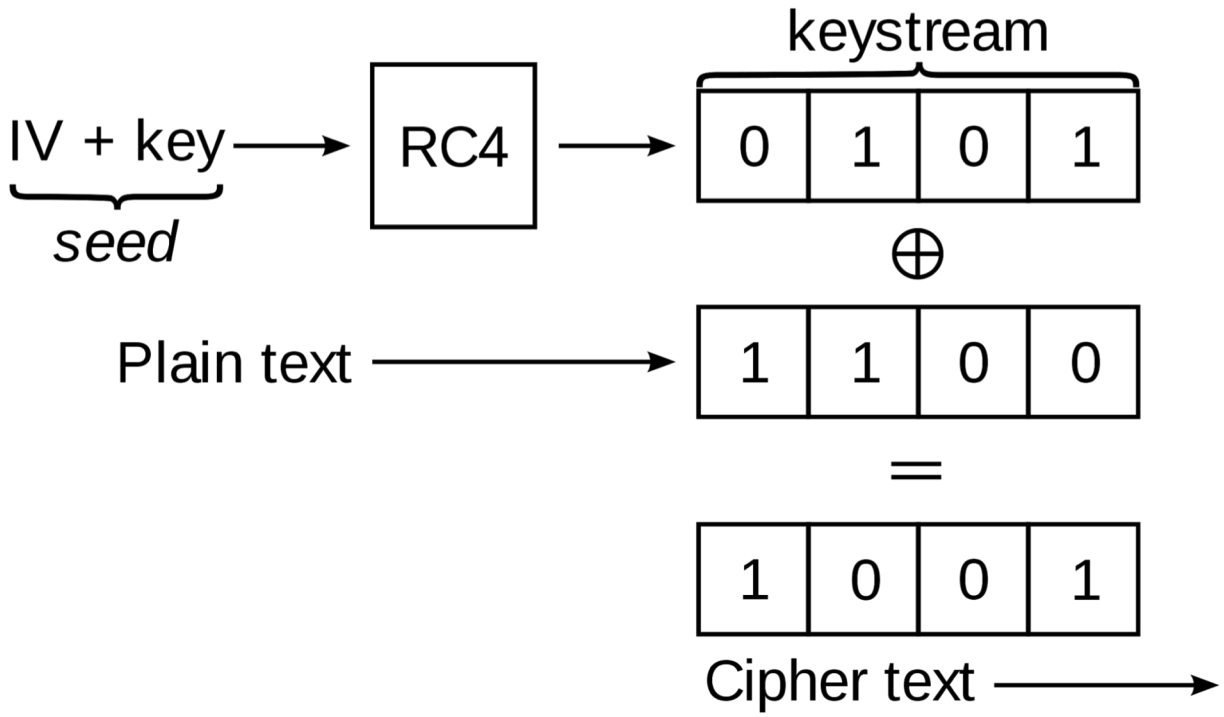
WEP Deprecated

- It was superseded by WPA, announced by WIFI Alliance in 2003.
- It was declared deprecated by IEEE in 2004.

According to https://wigle.net/stats#, 5.96% wireless network still uses WEP for its encryption today (June 2019), because there are some legacy devices that could only operate in some certain environment.



RC4 Algorithm - Overview



RC4 is a stream cipher, which works like a one time padding.

If Pseudo-Random Generator cannot guarantee randomness, key will be recovered.

First, let's see how RC4 works in detail.

Picture: https://en.wikipedia.org/wiki/Wired_Equivalent_Privacy

RC4 Algorithm - KSA (Key Schedule Algorithm)

```python
# Initialize S-box.
def initSBox(box):
    if len(box) == 0:
        for i in range(256):
            box.append(i)
    else:
        for i in range(256):
            box[i] = i

# Key schedule Algorithm (KSA) for key whose value is in unicode.
def ksa(key, box):
    j = 0
    for i in range(256):
        j = (j + box[i] + ord(key[i % len(key)])) % 256
        swapValueByIndex(box, i, j)
```

S-Box (substitution box) is a component widely used in symmetric key algorithm. User input a value and a seemingly irrelevant value is output.

I implemented it as an array with length 256 (permutation of all possible 1 byte).

RC4 Algorithm - KSA (Key Schedule Algorithm)

```python
# Initialize S-box.
def initSBox(box):
    if len(box) == 0:
        for i in range(256):
            box.append(i)
    else:
        for i in range(256):
            box[i] = i

# Key schedule Algorithm (KSA) for key whose value is in unicode.
def ksa(key, box):
    j = 0
    for i in range(256):
        j = (j + box[i] + ord(key[i % len(key)])) % 256
        swapValueByIndex(box, i, j)
```

The value of S-Box is disrupted in each loop by the user input key. There will always two index's values be swapped. This makes it very difficult to recover its original status.

Rc4 Algorithm - PRGA (Pseudo Random Generation Algorithm)

```python
# Pseudo-random generation algorithm (PRGA).
def prga(plain, box, keyStream, output):
    i = 0
    j = 0
    # Loop through every byte in plain text.
    for i in range(len(plain)):
        i = (i + 1) % 256
        j = (j + box[i]) % 256
        swapValueByIndex(box, i, j)
        keyStreamByte = box[(box[i] + box[j]) % 256]
        outputByte = chr(ord(plain[i-1]) ^ keyStreamByte)
        keyStream += chr(keyStreamByte)
        output += outputByte
    return (keyStream, output)
```

When generating the pseudo random key stream, two indexes' value will also be swapped, making each state different and hard to track its original state.

RC4 Weakness

```
sessionKey = iv + key
```

As a stream cipher, it is dangerous to use the same key over and over again, because the same key stream will be generated if the key is the same.

So, in WEP, apart from user input's key, an initialization vector is prepended to it, making each stream cipher different from others.

But, there are still weakness.

RC4 Weakness

```
12  def Key_Scheduling_Algorithm():
13      for i in range(256):
14          s.append(i)
15      j = 0
16      for i in range(256):
17          j = (j + s[i] + key[i % len(key)]) % 256
18          swap(s[i], s[j])
19
20  def Pseudo_Random_Generation_Algorithm():
21      i = 0
22      j = 0
23      for i in range(len(plain)):
24          i = (i + 1) % 256
25          j = (j + s[i]) % 256
26          swap(s[i], s[j])
27          keyStreamByte = s[(s[i] + s[j]) % 256]
28          outputByte = plain[i-1] ^ keyStreamByte
29      return (keyStreamByte, outputByte)
30
```

RC4 uses a constantly-changing S-Box to ensure randomness.

In the attack, FMS defined a Resolved Condition:

i = 1

j = s[1],

Y = s[j + s[j]] = keyStreamByte[0] = s[s[1] + s[j]] , will not be disturbed in future swap.

Probability greater than $e^{-3} \approx 0.05$.

Recover the key in WEP

First, let's define that after iterated over t times, the S-Box is at state S_t , the index are i_t and j_t , the current output key stream byte is Z_t . Then we have:

$z_t = S_t[S_t[i_t] + S_t[j_t]]$

We define L to be length of session key, and I be length of IV (In our case I = 3). Then we have key:

K = [IV[0], IV[1], IV[2], SK[0], SK[1],....SK[l - 1 - i]] (SK is user-entered secret key)

We attempt to recover the key starting from the first unknown secret key index A, this index locates at (A+3) in key.

As eavesdropper, we know the IV, encrypted cipher stream, and first byte of plain text 'aa', which is SNAP header used in WEP.

Recover the key in WEP

We want to examine the IV with a form of (A+3, N-1=255, V). V could be any number. We can see the reason why such form below:

In first round of ksa, the S-Box status is:

$A+3$	$N-1$	V	$K[3]$		$K[A+3]$	
0	1	2			$A+3$	
$A+3$	1	2			0	
i_0					j_0	

Recover the key in WEP

In the second round of ksa, the S-Box status is:

$A+3$	$N-1$	V	$K[3]$		$K[A+3]$	
0	1	2			$A+3$	
$A+3$	0	2			1	
	i_1				j_1	

Then, in the next round, j will be changed by adding V + 2.
All the rounds later can be calculated by attacker until he encountered the first byte he doesn't know (A+3). At this point, he can drop this IV trial by checking if S[0] and S[1] has changed.

Recover the key in WEP

If it has not changed, then the attacker knows that j will be added by $S_{A+2}[i_{A+3}]$ + K[A + 3], and then perform the swap. The S-Box status would be:

$A+3$	$N-1$	V	$K[3]$		$K[A+3]$	
0	1	2			$A+3$	
$A+3$	0	$S[2]$			$S_{A+3}[A+3]$	
					i_{A+3}	

Because the attacker knows S_{A+2} and j_{A+2}, if the attacker knows the value of $S_{A+3}[A + 3]$, he knows its location in S_{A+2}, which is the value of j_{A+3}, and hence he would be able to compute K[A + 3].

Recover the key in WEP

The formula is:

$$z0 = S[S[1] + S[S[1]]]$$

$$= S[0 + S[0]]$$

$$= S[A + 3]$$

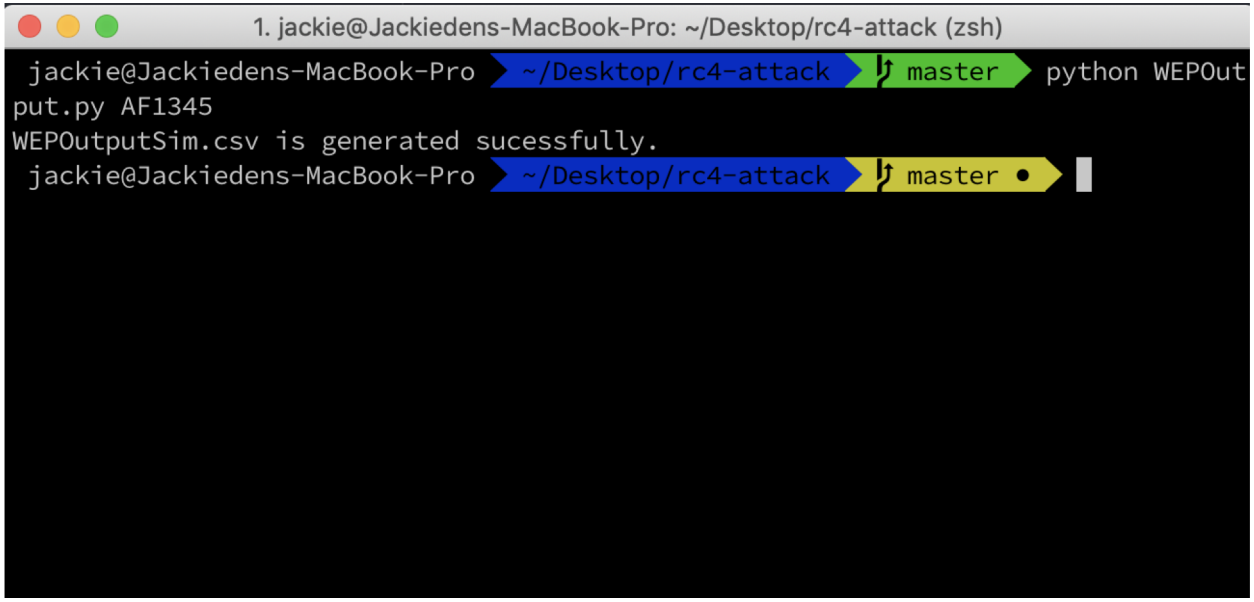
$$= S_A+2[j_{A+3}]$$

$$= S_A+2[j_{A+2} + S_{A+2}[A + 3] + K[A + 3]]$$

Demo of FMS Attack - Generate simulated WEP Packets

In the terminal, we run WEPOutput.py with user input key, for example, ‘AF1423’

```
$ python WEPOutput.py AF1423
```

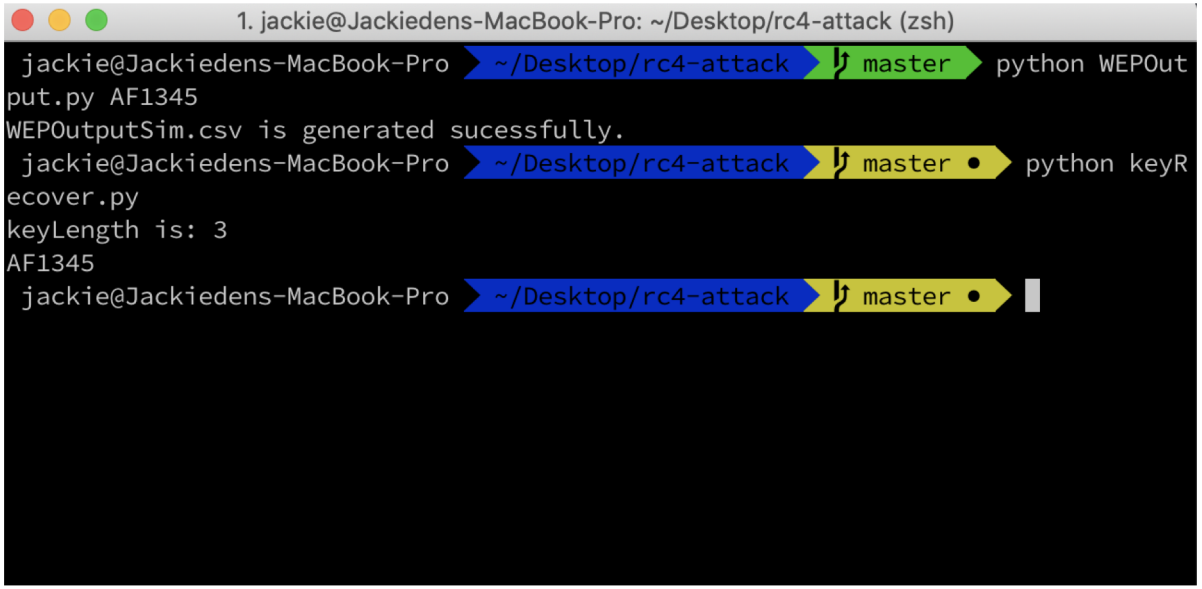


to create WEPOutputSim.csv files, which stores each possible IV in our desired form:
(A+3, 255, V).
And we also store the first key stream byte. We assume that these info is available to the attacker.

Demo of FMS Attack - Recover the key based on Packets

Then, we run keyRecover.py (this program utilizes WEPOutputSim.csv):

```
$ python keyRecover.py
```



It will run based on FMS Attack, and output the highest possible key. In our case, it matches the original entered key.

FMS Attack - Summary

In WEP, the IV is always prepended to the user input key. Since user input key is most likely remains the same for a same network, the attacker is able to eavesdrop a lot of packets and based on the known IV and SNAP header, the secret key can be recovered.

Thus, don’t use WEP encryption anymore!

Please note that a lot of graphs I use in this presentation are from FMS Attack source paper:

Fluhrer, S., Mantin, I., and A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4", Selected Areas of Cryptography: SAC 2001, Lecture Notes in Computer Science Vol. 2259, pp 1-24, 2001.