Diploma thesis Fachgebiet Theoretische Informatik

Summer term 2007

Fachbereich Informatik

TU Darmstadt

Attacks on the WEP protocol

Erik Tews

e_tews@cdc.informatik.tu-darmstadt.de

Supervisor: Prof. Dr. Dr. h. c. Johannes Buchmann

December 15, 2007

Abstract

WEP is a protocol for securing wireless networks. In the past years, many attacks on WEP have been published, totally breaking WEP's security. This thesis summarizes all major attacks on WEP. Additionally a new attack, the *PTW attack*, is introduced, which was partially developed by the author of this document. Some advanced versions of the *PTW attack* which are more suiteable in certain environments are described as well. Currently, the *PTW attack* is fastest publicly known key recovery attack against WEP protected networks.

Contents

1 Motivation

Since the IEEE 802.11 standard was released in its first version in 1997, IEEE 802.11 based wireless LANs (also called WLANs) quickly evolved to the most commonly used technology to wirelessly connect devices to an IP network. While the first release of the standard only allowed a transmission rate of 2 MBit, newer versions of the standard allowed transmission rates of 11 MBit (IEEE 802.11b) or 54 MBit (IEEE 802.11a and IEEE 802.11g). With IEEE 802.11n, which was only available as a draft at the moment this document was written, this will even be raised to 300 MBit bandwidth, which is sufficient for high definition video content and fast file transfers.

A wireless LAN usually consists of at least one base station called *access point* and one or more wireless clients connected to these base stations. The base stations can be interconnected using wired links or wireless links and be connected to another wired network. This kind of network is usually called *infrastructure mode*. Another more seldom used mode is the so called *ad hoc mode*, in which no base stations are used and all clients communicate directly.

Wireless LANs can be found nearly everywhere today. Most mobile computers ship with built-in wireless LAN hardware by default and most other computers can be equipped with additional hardware. Even some mobile phones and PDAs ship with wireless LAN hardware or can be upgraded. Besides that, wireless LAN is used in some industrial applications like *point of sale terminals* and *info screen displays*.

Most people use wireless LANs to connect all devices in their home network to a single wire based internet connection. Universities allow their students to connect their mobile computers to their network and use their internet connection on campus. In popular public places like train stations or restaurants, companies sell internet access over their wireless LANs, which are also known as *hot spots*. Enterprises are using wireless LANs to connect their workers and sometimes visitors to their company network.

Because all data is transmitted wirelessly, extra security is needed in these networks. Without, an attacker could read all wireless traffic or use the network against the network operators will. This also was a concern to the creators of IEEE 802.11 standard, who designed a simple protocol called WEP which stands for *Wired Equivalent Privacy* and which should provide the same level of privacy to the users of IEEE 802.11 based wireless networks as they would have on a wired network.

In a WEP network, all stations share a single secret key, the so called *root key*. Every time a station in the network sends data, a so called *per packet key* is derived from the *root key* and used as a key for the RC4 stream cipher [Riv92] to generate a key stream. An additional checksum is appended to the packet and the packet then is XORed with the key stream and send. At the first look, this protocol seems to be a good choice for a small network. Because sharing a single secret key with all employees and keeping is still secret can be difficult for an enterprise, modified versions of this protocol have been developed, which allow other authentication methods like username/password or smartcard based authentication.

Unfortunately, the WEP protocol has some serious design flaws. Four years after the release of the first version of IEEE 802.11, in 2001, some cryptographic researchers showed [FMS01] that the secret key of such a network can be revealed within hours and full access to the network is possible for an attacker. Because the protocol has no kind of *perfect forward secrecy* ([Men01] page 496), the attacker can also decrypt previously captured traffic.

While for the first key recovery attack against WEP fixes where proposed, by modifying the protocol slightly, without breaking interoperability with older stations, more advanced attacks started to appear. Soon it became clear that a redesign of the protocol was absolutely necessary. In 2004, the final version of the IEEE 802.11i standard was released which defines the successor protocol for WEP which is mostly known as WPA or WPA2.

While WPA or WPA2 seems to be a secure protocol with no known design flaws, WEP is still used and some vendors still ship devices which can only connect to unsecured or WEP networks. In 2006, a German student estimated that about 61% of all networks in a larger area in Germany still use WEP and 22% use no protection at all. In total, there could be about 5,000,000 networks in Germany which still use WEP. [Dör06]

Currently, weaknesses in WEP are actively exploited. In 2007, newspapers reported [BO07] that crackers gained access to a company's private network and stole the customer records including credit card data of about 45,000,000 customers.

In 2007, Ralf-Philipp Weinmann, Andrei Pyshkin and I started looking at the current attacks on the WEP protocol and looked for improvements. As a part of our results, a new attack on the WEP protocol was developed [TWP07] which is able to recover the secret key of an WEP protected network a magnitude faster than all previous attacks.

1.1 Structure of this document

The structure of this document is as follows: In Chapter 2, the notation and the definition of certain special terms is explained. Chapter 3 gives an introduction to the RC4 stream cipher. Chapter 4 gives an overview of IEEE 802.11 and WEP. In Chapter 5, attacks on WEP unrelated to RC4 are described, while Chapter 6 describes general attack on RC4. Chapters 7 and 8 contain new attacks on RC4, which are partially the result of my research. Chapter 9 describes WPA, the successor protocol to WEP which prevents all of these attacks. Chapter 10 contains the conclusion. Chapter 11 lists all contributions to this document.

2 Notation and special words

First of all, a common notation for all attacks is needed.

2.1 Mathematical notation

Numbers in this document are usually written in decimal. For example 13 is the number thirteen. In some cases, most times when it comes to values in headers of certain data packets, hexadecimal notation is used. In this case, numbers are written in a bold style; for example **1A** is the number *twenty six*.

The signs $+$, $-$, $\cdot$ are the signs for addition, subtraction and multiplication. $(\mathbb{Z}/n\mathbb{Z})^+$ is the additive group of the numbers 0 to $n-1$, where all additions are done $\bmod n$. When an operation like $c = a+b$ is done in $(\mathbb{Z}/n\mathbb{Z})^+$, we write $c \equiv_n a + b$ or $c = a + b \bmod n$. In some parts of this document, all operations are done in $(\mathbb{Z}/n\mathbb{Z})^+$. If this is the case, it is announced at the beginning of the section and just $c = a + b$ is written.

For arrays, the $[\cdot]$ notation is used, like it is used in many popular programming languages like *C* or *Java*. All array indices start at 0. For permutations, the same notation is used. For example if P is the identity permutation, $i = P[i]$ holds for every value of i. P^{-1} is the inverse permutation of P. If $\mathbb{F}$ is a finite field, $\mathbb{F}[X]$ is the set of polynomials over $\mathbb{F}$.

If a has a numeric value which is close to b, $a \approx b$ is written.

For sets, the $\{\cdot\}$ notation is used. For example, if A is the set consisting of the values a_1, a_2 and a_3, $A = \{a_1, a_2, a_3\}$ is written. If a value a is randomly chosen from a set A using a *uniform distribution*, $a \leftarrow_R A$ is written.

2.2 Complexity theory

In this document, a system with a mostly fixed key length is examined. This makes it hard to use complexity theory to describe the security of this system. However, I will use some terms from the area of complexity theory with an adapted meaning.

An attack on a cryptosystem is *efficient*, if it can be executed much faster on a system than an exhaustive search for the correct key. The computational effort of an attack is *negligible*, if it can be performed in some seconds to minutes on an average computer sold in the year 2007.

2.3 Oracles

Usually, an oracle is a black box which is able to perform certain operations which an attacker cannot perform himself. The oracle can only be accessed using a defined interface and an attacker cannot see any of the internals of the oracle, besides what he can access over the interface. For example, an oracle could be in the possession of a secret key and have an interface which accepts a plaintext and returns the corresponding ciphertext. An attacker who has access to this oracle can now encrypt arbitrary plaintexts, but cannot ask the oracle for its secret key or for the decryption of a ciphertext.

2.4 Special notation

Later in Chapter 3, a stream cipher called *RC4* is introduced. A special notation for the internal state of the RC4 stream cipher is introduced in this chapter too.

3 The RC4 stream cipher

RC4 [Riv92] is a often used stream cipher designed by Ron Rivest in 1987. RC4 was kept as a trade secret by RSA Data Security until it leaked out in 1994 [Ste94]. Today, RC4 is used in the SSL/TLS protocol, the WEP protocol and its successor, the TKIP protocol as in many other protocols and applications.

3.1 An overview over the RC4 stream cipher

RC4 consists of two algorithms. The *RC4 Key Scheduling Algorithm* (RC4-KSA) transfers a key K of length 1 to 256 bytes [1] into an internal state of RC4. The internal state of RC4 consists of an array S describing a permutation of the numbers from 0 to 255 and two integers i and j with $0 \leq i, j \leq 255$ used as pointers to elements of S. After the internal state has been initialized, the *RC4 Pseudo Random Generator Algorithm* (RC4-PRGA) can be used to generate a key stream of arbitrary length. With every output byte produced by the RC4-PRGA, the internal state of RC4 is updated.

The function *swap(*S*,a,b)* swaps the elements $\mathsf{S}[a]$ and $\mathsf{S}[b]$ in the array S.

Listing 3.1: RC4-KSA

```
for i ← 0 to 255 do
  S[i] ← i
end
j ← 0
for i ← 0 to 255 do
  j ← j+S[i]+K[i mod len(K)] mod 256
  swap(S, i, j)
end
i ← 0
j ← 0
```

As you can see, RC4 has an interesting design. It just uses 8 bit additions and bytewise random memory access on a 256 byte memory region. Therefore, it can even be used on very restricted CPUs like 8 bit processors. Additionally, RC4 is very fast on modern 32 and 64 bit CPUs outperforming many block ciphers. For example, RC4 encrypts data with only 7.5 clock cycles per byte on

[1] Some documents specify, that an RC4 key has to have at least 5 bytes. Technically, RC4 can work with shorter keys, making it totally insecure because the key space is too small.

Listing 3.2: RC4-PRGA

```
i ← i + 1 mod 256
j ← j + S[i] mod 256
swap(S, i, j)
return S[ S[i] + S[j] mod 256 ]
```

a Pentium-M 1.7 GHz CPU compared to 23.6 clock cycles per byte for AES128 in CBC mode.

On the other side, RC4 maintains an internal state of at least $\log_2(256! \cdot 256^2) \approx 213$ bytes, making it unusable in situations where memory is too restricted. Efficient implementations use at least 258 bytes for the internal state. Additionally, every state update, which happens with every output byte, might affect the next output byte, making RC4 hard to parallelize and slowing it down on CPUs with long pipelines.

3.2 Analyzing the RC4 stream cipher

For analyzing the RC4 stream cipher, it is often useful to look at some modified versions of the algorithm.

3.2.1 The generalized RC4 stream cipher

The *generalized RC4 stream cipher* was used by Klein [Kle06] in his analysis. The original RC4 algorithm works on elements of the group $(\mathbb{Z}/256\mathbb{Z})^+$. Klein generalizes this idea to arbitrary groups $(\mathbb{Z}/n\mathbb{Z})^+$, where n is an arbitrary natural number, most times used as a kind of complexity parameter. Please note that this modified algorithm is usually not used in implementations, it just helps analyzing the unmodified version of RC4. If you set n to 256, you get the original algorithm again.

Listing 3.3: Generalized RC4-KSA

```
for i ← 0 to n−1 do
  S[i] ← i
end
j ← 0
for i ← 0 to n−1 do
  j ← j+S[i]+K[i mod len(K)] mod n
  swap(S, i, j)
end
i ← 0
j ← 0
```

Listing 3.4: Generalized RC4-PRGA

```
i ← i + 1 mod n
j ← j + S[i] mod n
swap(S, i, j)
return S[ S[i] + S[j] mod n ]
```

We will call a single iteration of the loop between lines 5 and 8 a *step* of the RC4-KSA and a the generation of a single word of output by the RC4-PRGA a *step* of the RC4-PRGA.

3.2.2 The generalized randomized RC4 stream cipher

In RC4, the variable j seems to change randomly. This observation can be idealized to the idea of the *generalized randomized RC4 stream cipher*.

Listing 3.5: Generalized randomized RC4-KSA

```
for i ← 0 to n−1 do
  S[i] ← i
end
j ← 0
for i ← 0 to n−1 do
  j ← RND(n)
  swap(S, i, j)
end
i ← 0
j ← 0
```

Listing 3.6: Generalized randomized RC4-PRGA

```
i ← i + 1 mod n
j ← RND(n)
swap(S, i, j)
return S[ S[i] + S[j] mod n ]
```

$RND(n)$ is a randomized function returning independent values from 0 to $n-1$ inclusively from a uniform distribution. Please note that the *generalized randomized RC4 stream cipher* is technically not a stream cipher anymore. Depending on what value is assigned to j, the algorithm is likely to produce different key streams from the same key.

Ilya Mironov was the first person I know of, who published [Mir02] this formalized randomized version of RC4.

3.2.3 Notation and visualization

The following notation is used to analyze what happens in certain steps of the KSA or PRGA.

S_k is the content of S after exactly k swaps have been done on S during the KSA or PRGA. Trivial swaps, where an element is exchanged with itself, are counted too. S_0 is therefore the identity permutation. j_k respectively is the content of j after k swaps on S. This notation can be used on all three versions of the RC4 algorithm we introduced.

K is a key and X is a key stream.

Sometimes the need the first l bytes of output of the RC4-PRGA, initialized with the key K. This is the output of the function $\mathrm{RC4}(\mathsf{K}, l)$.

To make RC4 easier to understand, the following visualization is used. Most times, an attacker is not interested in all values of S, instead the attacker is mostly interested in the values which are modified in certain steps of the RC4-KSA, and in the values which are used to generate the output in the RC4-PRGA. For example, if the key $\mathsf{K} = 23, 42, 232, 11$ is being used and we are looking at the first 4 steps of the RC4-KSA, we are interested in $\mathsf{S}[0], \mathsf{S}[1], \mathsf{S}[2], \mathsf{S}[3], \mathsf{S}[4], \mathsf{S}[23], \mathsf{S}[44], \mathsf{S}[58]$, $\mathsf{S}[66]$ and $\mathsf{S}[85]$. These steps are illustrated in figure 3.1.

After the rest of the RC4-KSA, the following 3 bytes of output are generated: $85, 161, 104$. These steps are illustrated in figure 3.2.

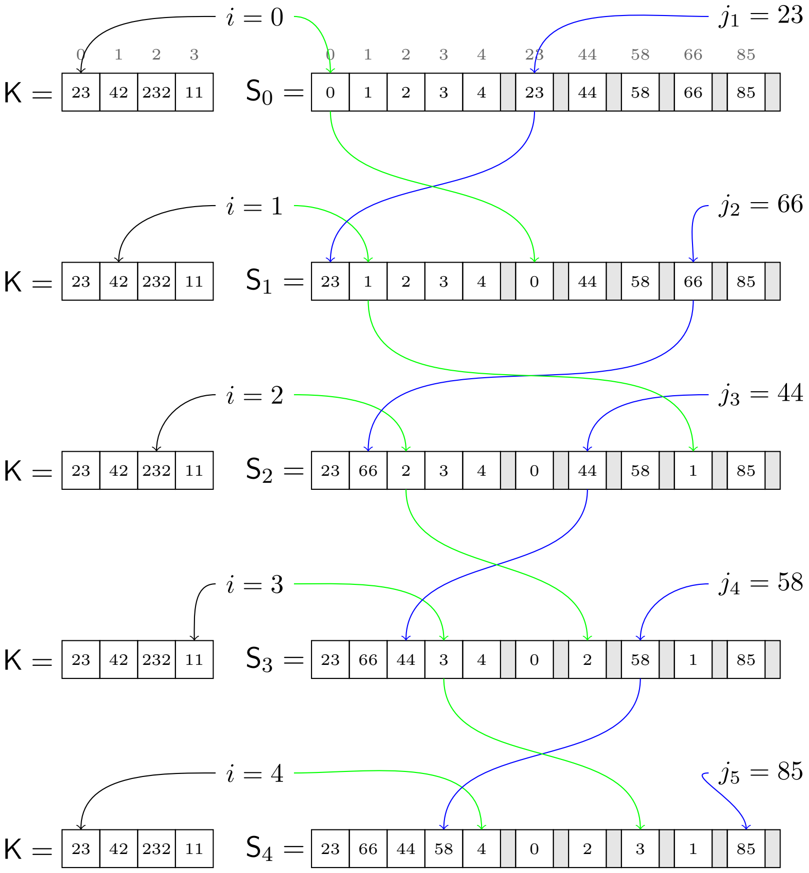


Figure 3.1: First 4 steps of RC4-KSA for $\mathsf{K} = 23, 42, 232, 11$

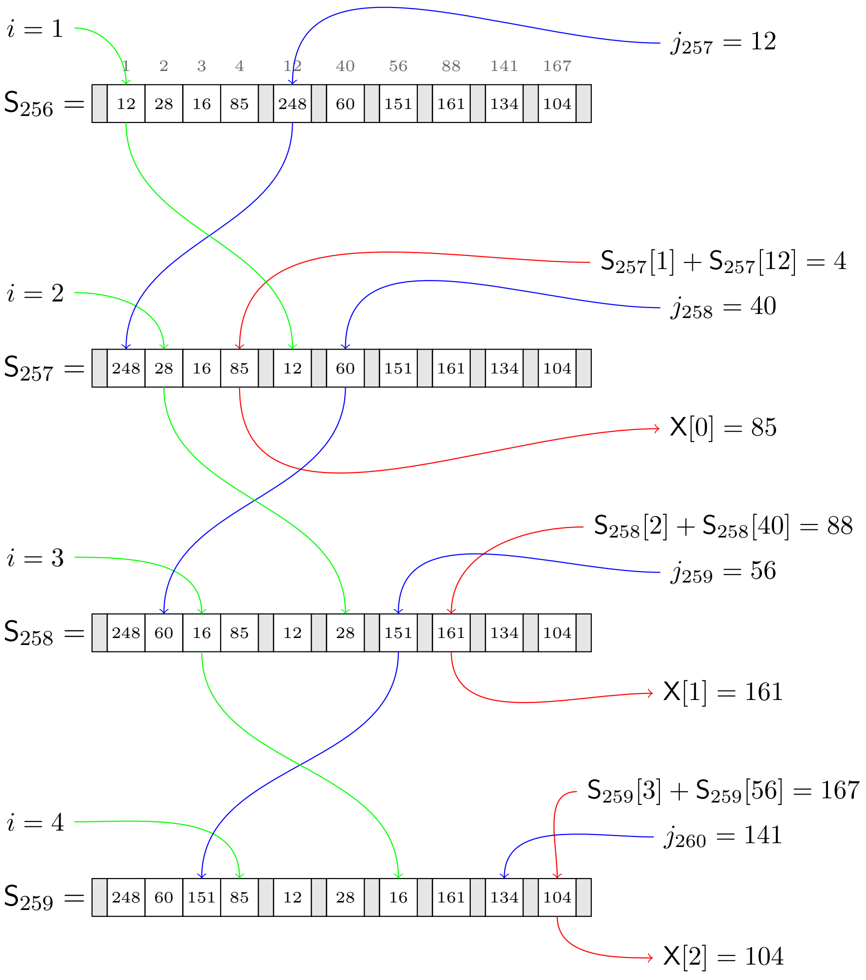


Figure 3.2: First 3 bytes of output of RC4-PRGA for $\mathsf{K} = 23, 42, 232, 11$

4 IEEE 802.11 and WEP

In 1997, the IEEE released the first version of the IEEE 802.11 standard for wireless networking. Even before 1997, it was possible to connect computers over a wireless connection:

- A lot of devices are equipped with an infrared port, which allows infrared communication between two devices over a distance of half a meter or less.
- A lot of wireless phones use the DECT protocol to communicate with their base station. Beside the phones, some vendors also sell DECT wireless modems, which allow wireless modem connections. The range of the system is limited to a few hundred meters under good conditions.
- GSM was the most common standard for mobile phones in Germany in 1997. Beside voice communication, GSM allows data communication so that a laptop computer can be connected to another system using a modem connection over GSM. The distance between a mobile computer with a GSM modem and the next GSM base station can be several kilometers.
- The ham radio network allows data connections between two hams using a special transmitter and a computer. The range of such connections can exceed the maximum range of a GSM mobile phone. To use this network, a special license has to be obtained.

4.1 IEEE 802.11 (1997)

In general, the speed of these connections was much slower than 2 MBit and not suitable for larger data transfers. IEEE 802.11 defined a new protocol to network computers without a wire with a maximum speed of 2 MBit. The IEEE 802.11 standard allows the transmission of the signal at a frequency band at about 2.4 GHz or over infrared. In practice, the infrared option is never used. The range of such connections is usually limited to a few hundred meters at most, but can be extended to multiple kilometers using special hardware. Today, the original IEEE 802.11 standard is obsolete and is not used anymore.

In addition, the original IEEE 802.11 document specifies a simple security protocol called *Wired Equivalent Privacy (WEP)*, which should provide the same level of privacy to the legitimate users of an IEEE 802.11 network, as they would have with a wired network. In most scenarios, the range of the network cannot be fine controlled and an attacker, who is in the same building or next to the building, is able to capture all of the traffic on the network. To prevent unauthorized access to the network by an attacker, all data frames are integrity protected and encrypted before they are send, if WEP is being used.

In the following years, enhancements of the IEEE 802.11 standard were released. We will only focus on a few ones, which are of interest for this diploma thesis.

4.1.1 IEEE 802.11b (1999)

In 1999, IEEE 802.11b was released. IEEE 802.11b did not change anything related to WEP, but the maximum bandwidth was increased to 11 MBit. This is of interest for us, because an attacker can now send and collect data packets faster. IEEE 802.11b is backward compatible with the original IEEE 802.11 standard.

Today, IEEE 802.11b hardware is still sold and used, but most new products support IEEE 802.11g or IEEE 802.11n, which allows faster transfer rates.

4.1.2 IEEE 802.11a (1999)

In 1999, IEEE 802.11a was released too. As IEEE 802.11b, IEEE 802.11a does not introduce any new security features, but allows transmitting data in a 5 GHz frequency band and allows a maximum bandwidth of 54 MBit. Here, an attacker would be able to collect data faster than in an IEEE 802.11b based network.

IEEE 802.11a is not as popular as other versions of the IEEE 802.11 standard family, which operate at 2.4 GHz, but is integrated in some upper class wireless cards or laptop computers. No other standard has been released until now, which offers more bandwidth at the 5 GHz band. Most wireless equipment sold today which supports IEEE 802.11a additionally supports IEEE 802.11g or IEEE 802.11b, some even supports a draft version of IEEE 802.11n.

4.1.3 IEEE 802.11g (2003)

In 2003, IEEE 802.11g was released, which offers 54 MBit bandwidth on the 2.4 GHz frequency band, as IEEE 802.11a does in the 5 GHz frequency band. As all other standards before, IEEE 802.11g does not improve the security. Here, an attacker will be able to capture traffic at basically the same speed as on IEEE 802.11a.

IEEE 802.11g can be seen as the most popular standard for wireless LAN today. Nearly all new wireless cards or laptop computers today support IEEE 802.11g. IEEE 802.11g is backward compatible with IEEE 802.11b, so that a seamless migration from IEEE 802.11b is possible.

4.1.4 IEEE 802.11n (draft, unreleased)

Currently, a draft version of IEEE 802.11n is available, which offers 300 MBit of bandwidth to its users. IEEE 802.11n based networking cards operate in the 2.4 GHz frequency band and are backwards compatible to IEEE 802.11g. Vendors have already begun to sell IEEE 802.11n networking cards based on a draft version of this standard.

IEEE 802.11n does not have as high a market share as IEEE 802.11g has and at the moment, IEEE 802.11n hardware is more expensive than IEEE 802.11g based hardware. Because applications like high definition video and faster internet connections demand a higher bandwidth than IEEE 802.11g offers, this can be expected to become the next most popular standard for wireless networking.

4.1.5 Proprietary vendor extensions

Because the development of IEEE 802.11n is still not finished, some vendors have started to implement their own enhancements of IEEE 802.11b or IEEE 802.11g. Some cards are sold which offer 22, 108 or 125 MBit of bandwidth, which usually means that these cards additionally support a vendor specific protocol, which allows faster transfer rates than IEEE 802.11b or IEEE 802.11g. Usually, all of these cards are able to communicate with other IEEE 802.11b or IEEE 802.11g based hardware at the maximum speed IEEE 802.11b or IEEE 802.11g offers.

Additional remarks

Somebody might wonder why the year 1997, 1999 or 2007 is written on the IEEE 802.11 standard. This is due to the fact that the IEEE releases updates of their standards after the first final version has been released. After a new version has been released, the status of the older versions is changed to *archived*.

The data rates in these standards must be seen as physical data rates. Due to protocol overhead, only about 50% of the bandwidth is available to the payload of the transmissions.

4.2 General structure of an IEEE 802.11 based wireless LAN

A IEEE 802.11 based network is usually identified by a name, called *ESSID* in the terminology of IEEE 802.11. This is a short string, mostly the name of the operator or manufacturer of the network or the purpose of the network. For example *HotelNetwork*, *PublicWLAN* or *Dlink* could be valid values for an

ESSID. There are two types of networks defined in IEEE 802.11: The *infrastructure network* and the *ad hoc network*.

In an *ad hoc network*, all stations (STA) communicate directly with each other, without any kind of central component. This is used seldom, to connect stations for a short time where no central infrastructure is available. Sending a song from one portable MP3 player to another one could be such a situation. Figure 4.1 shows an example of such a network.

The second type is called *infrastructure network*. The *infrastructure mode* is the most common mode to operate a IEEE 802.11 wireless LAN. For the rest of this thesis, we will only focus in *infrastructure networks*, but most of the results can be applied to *ad-hoc networks* too. A *basic service set* (BSS) is a station, which acts as a base station for other stations. If this station provides access to a local network, this station is called *access point* (AP). For the rest of this work, I will assume that a BSS is always an AP and only use the word AP for these kind of stations. All access points are interconnected, but the area covered by the access points is allowed to be disjoint. I will call a station in an *infrastructure network* which is not an AP a *client*.

Every *access point* is identified by an BSSID, an 48 bit numerical value, usually set by the vendor as a hardware address for an Ethernet card. In general, I will name these addresses *MAC addresses*, no matter if the station is an AP or a client. Every access point broadcasts his BSSID and ESSID in intervals of mostly 100 ms. A client who wants to become a member of a network has to *associate* with an access point using a special handshake.

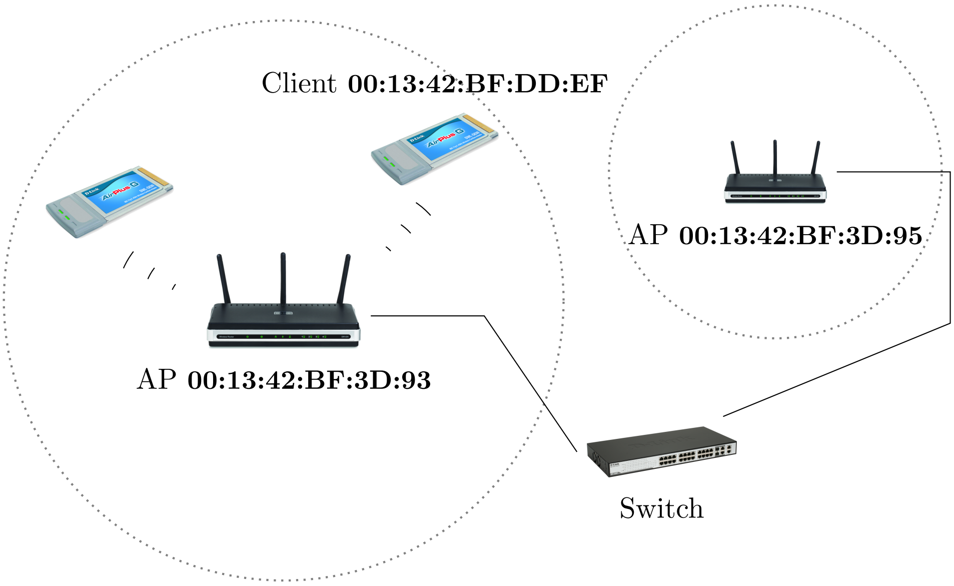


Figure 4.1: An example infrastructure network

Sometimes, network operators disable the broadcasting of the ESSID in their access point as a kind of security feature. This feature is known as *hidden network*. During the handshake, a client must send the ESSID of the network he wants to join. The idea behind is, that people hope that an attacker will not be able to join the network, because he does not know the networks ESSID. Of course this does not provide real security, because an attacker can just sniff the handshake of another client, read the ESSID there from and then use it for himself.

Most home networks only consist of a single AP and a single client, while other networks which span a whole university might consist of hundred of APs and thousand of clients.

4.3 WEP

The *Wired Equivalent Privacy* protocol, or short *WEP* protocol, is described in the IEEE 802.11 standard. In addition, the IEEE 802.11i standard contains a description of the protocol. We will later have a look at IEEE 802.11i.

In most networks, a single secret key, called *root key* (Rk) is shared between all stations. The WEP protocol allows up to 4 or in special cases even more different keys, but most network operators just use a single secret key. If a network is WEP protected, it is announced in the beacon frame. While the first version of the WEP standard only allowed a 40 bit root key, further versions allowed 104 bit root keys too. Some vendors additionally implemented longer root key with a length up to 232 bit.

4.3.1 Data encryption and integrity protection

Every data frame send by a station in a WEP protected network is encrypted an integrity protected. Non-data frames, like beacon frames, acknowledgment frames and similar frames are not protected by WEP at all. When a station sends a packet, the following steps are executed.

1. The station picks a 24 bit value called initialization vector IV. We will later use this value bytewise and write $\mathsf{IV}[0]||\mathsf{IV}[1]||\mathsf{IV}[2]$ for it. The IEEE 802.11 standard does not specify how to choose this value. Beside some minor modifications, most vendors implemented one of the following two methods:
   - The IV is chosen by a pseudo random number generator PRNG independently from all other packets send by this station.
   - The station always remembers the last IV used. When a new IV needs to be chosen, the station interprets the last IV used as a number and adds 1 to this number. When the highest possible number is

reached, the station starts again with 0. On startup the IV counter either takes a fixed value or a random number is assigned to it.

2. The IV is prepended to the *root key* and form the *per packet key* $\mathsf{K} = \mathsf{IV}||\mathsf{Rk}$.
3. A CRC32 checksum of the payload is produced and appended to the payload. This checksum is called *Integrity Check Value* (ICV).
4. The *per packet key* K is feed into the RC4 stream cipher to produce a key stream X of the length of the payload with checksum.
5. The plaintext with the checksum is XORed with the key stream and form the ciphertext of the packet.
6. The ciphertext, the initialization vector IV and some additional header fields are used to build a packet, which is now send to the receiver.

The whole process is visualized in figure 4.2. The sequence of operations can be different, for example the CRC32 value can be calculated independently of the key stream.

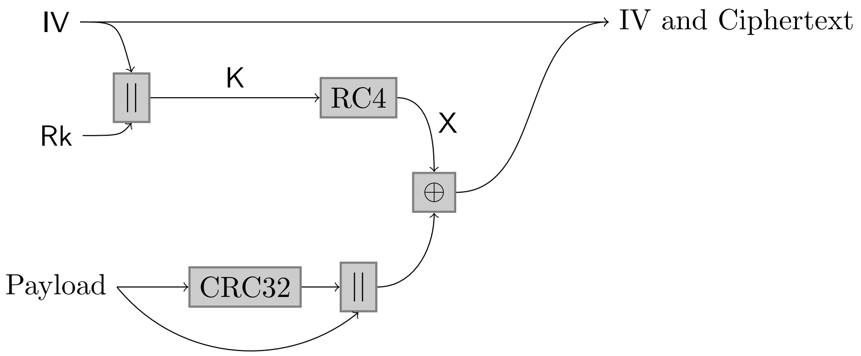


Figure 4.2: WEP encryption diagram

The packet being send now contains the following header fields:

Frame control contains general information about the frame (is it a data, management, or control frame. . .) and the transmission (has the station more packets to send. . .)

Duration, ID contains the expected duration of this transmission and some other values in special cases.

Address 1,2,3 contains the following addresses. The address of the AP the packet is send from/to, the address of the destination station and the address of the source station. In a special mode called *WDS*, where two APs communicate directly with each other, there is a fourth address, the address of the second AP.

Sequence control contains information about fragmentation. The IEEE 802.11 protocol is able to fragment packets before they are transmitted.

WEP parameters contains the IV which was used to encrypt this packet, and a *key index*. The *key index* is used to identify the correct key, when more than one key is used in a network.

Payload and ICV is the encrypted payload of the packet including a CRC32 checksum at the end of the payload which is called *Integrity protection value* (ICV). Payload and ICV are encrypted.

Frame control	Duration, ID	Addr. 1	Addr. 2	Addr. 3	Sequence control	WEP parameters	Payload, ICV
2 bytes	2 bytes	6 bytes	6 bytes	6 bytes	2 bytes	4 bytes	variable

Figure 4.3: IEEE 802.11 data frame format

The whole header is shown in figure 4.3. The CRC32 checksum in the ICV is only computed over the encrypted payload. There is no cryptographic protection for all unencrypted header fields.

4.3.2 Authentication

WEP additionally defines two modes how a station can authenticate itself before joining a network.

Open system authentication

In this mode, no authentication is done at all. A client just sends an request to the AP to be authenticated. The AP responds with *success*. None of these messages is encrypted. Figure 4.4 contains an illustration of these protocol steps.

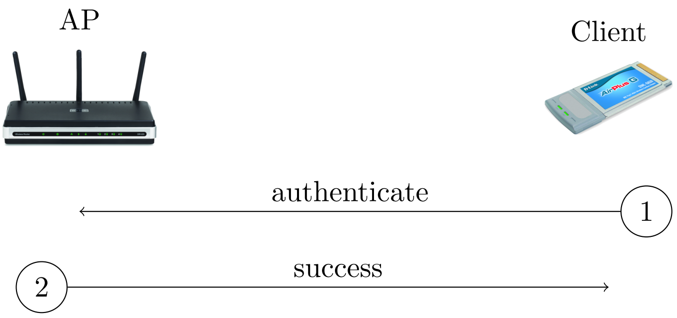


Figure 4.4: Open system authentication

Shared key authentication

In this mode, a challenge response handshake with 4 messages is used. In the first frame, the client asks the AP to join the network. The AP responds with a random number *rand*, which is transmitted in cleartext. The client now needs to send an encrypted frame containing *rand*. If this frame is decrypted correctly by the AP and contains *rand*, the access point allows the client to join the network. Figure 4.5 contains an illustration of these protocol steps.

The basic idea is that a client who is not in possession of the secret key will not be able to construct a valid third frame and will therefore not be able to join the network.

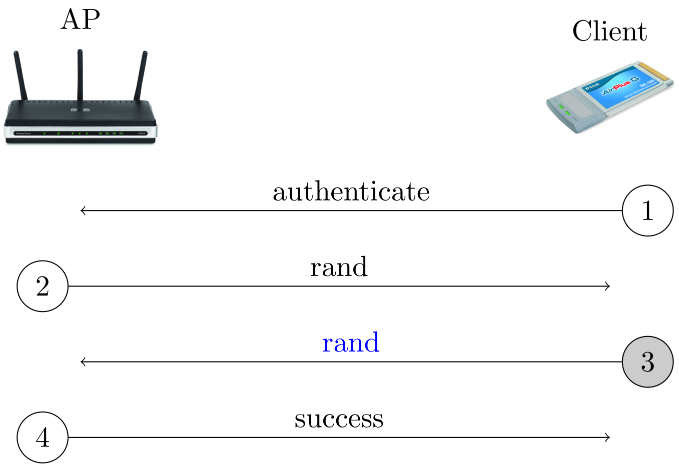


Figure 4.5: Shared key authentication

5 Previously known attacks on WEP not related to RC4

A number of attacks are known on WEP which are not based on a weakness of the RC4 stream cipher.

Most of these attacks and a lot of attacks based on weaknesses in the RC4 stream cipher are implemented in the *aircrack-ng toolsuite*. Aircrack-ng is available under the GPL license from http://www.aircrack-ng.org.

To benchmark CPU expensive attacks, two machines have been used. One is running on a quad-core *Intel(R) Xeon(R) CPU X3210* running at 2.13 GHz. The other machine is running on two *AMD Opteron(tm) Processor 2218* running at 2.6 GHz. Both machines got more than 2 GB of main memory, but all attacks required less than 200 MB of main memory.

5.1 Packet injection

Attack 1 *An attacker who has captured an encrypted packet in a WEP network, can later reinject this packet, and it will still be accepted by the network.*

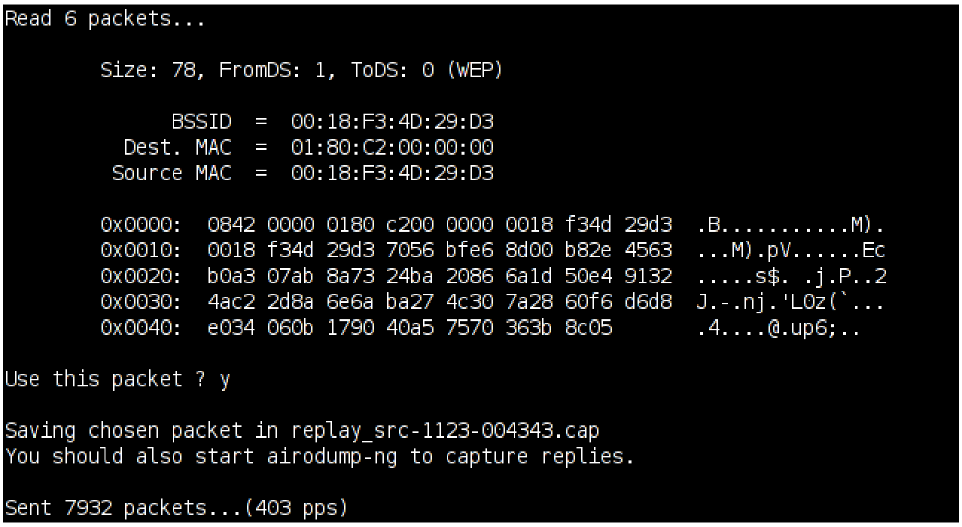
Packet injection is sometimes understood as not a real attack on WEP, because WEP was never designed to be resistant against such an attack. A packet sent in a WEP protected network which has been intercepted by an attacker, can later be injected into the network again, as long as the key has not been changed and the original sending station is still in the network. If the sending station is not in the network anymore, the senders (and the receivers) address can be changed to a station that is still in the network. This is possible, because these fields are not protected by the ICV.

5.1.1 Implementation

aircrack-ng contains various modes how packets can be replayed. To listen to the interface `wifi0` and wait for packets for BSSID `00:18:F3:4D:29:D3`, an attacker has to execute the following command:

```
./aireplay-ng -2 -b 00:18:F3:4D:29:D3 -h 00:14:6C:F7:17:0E wifi0
```

When a suitable packet for injection is received, the encrypted packet will be displayed and the attacker is prompted if he wants to use this packet. If the attacker answers with `y`, the station address `00:14:6C:F7:17:0E` will be used as a source address for the reinjected packets.



```
Read 6 packets...

        Size: 78, FromDS: 1, ToDS: 0 (WEP)

              BSSID  =  00:18:F3:4D:29:D3
          Dest. MAC  =  01:80:C2:00:00:00
         Source MAC  =  00:18:F3:4D:29:D3

        0x0000:  0842 0000 0180 c200 0000 0018 f34d 29d3  .B..........M).
        0x0010:  0018 f34d 29d3 7056 bfe6 8d00 b82e 4563  ...M).pV......Ec
        0x0020:  b0a3 07ab 8a73 24ba 2086 6a1d 50e4 9132  .....s$. .j.P..2
        0x0030:  4ac2 2d8a 6e6a ba27 4c30 7a28 60f6 d6d8  J.-.nj.'L0z(`...
        0x0040:  e034 060b 1790 40a5 7570 363b 8c05       .4....@.up6;..

Use this packet ? y

Saving chosen packet in replay_src-1123-004343.cap
You should also start airodump-ng to capture replies.

Sent 7932 packets...(403 pps)
```

Figure 5.1: aircrack-ng 1.0 beta 1 injection results

During the attack, an output similar to the one in figure 5.1 will be displayed.

Alternatively, an attacker can execute the command:

```
./aireplay-ng -3 -b 00:18:F3:4D:29:D3 -h 00:14:6C:F7:17:0E wifi0
```

Aircrack will try to find an encrypted *ARP packet* and use the first one found for injection. No user interaction is needed here.

5.2 Fake authentication

Attack 2 *Fake authentication: An attacker can join a WEP protected network, which supports* Open System authentication*, without knowing the secret* root key*. An attacker can join a WEP protected network, which support* Shared Key authentication*, if he has captured a full* Shared Key authentication handshake *between a station and an* access point*.*

The *fake authentication* attack on the WEP protocol allows an attacker to join a WEP protected network, even if the attacker has not got the secret *root key*.

IEEE 802.11 defines two ways a client can authenticate itself in an WEP protected environment. The first method is called *Open System authentication*.

Here, a client just sends a message to an access point, telling that he wants to join the network using *Open System authentication*. The access point will answer the request with *successful*, if he allows *Open System authentication*.

As you can see, the secret *root key* is never used during this handshake, allowing an attacker to perform this handshake too and to join an WEP protected network without knowledge of the secret *root key*.

The second method is called *Shared Key authentication*. *Shared Key authentication* uses the secret *root key* and a challenge-response authentication mechanism, which should make it more secure (at least in theory) than *Open System authentication*, which provides no kind of security.

> In a Shared Key authentication system, identity is demonstrated by knowledge of a shared, secret, WEP encryption key.[CWKS97]

First, a client sends a frame to an access point telling him, that he wants to join the network using *Shared Key authentication*. The access point answers with a frame containing a challenge, a random byte string. The client now answers with a frame containing this challenge which must be WEP encrypted. The access point decrypts the frame and if the decrypted challenge matches the challenge he send, then he answers with *successful* and the client is authenticated.

An attacker who is able to sniff an *Shared Key authentication* handshake can join the network itself. First note, that besides the APs challenge, all bytes in the third frame are constant and therefore known by an attacker. The challenge itself was transmitted in cleartext in frame number 2 and is therefore known by the attacker too. The attacker can now recover the key stream which was used by WEP to encrypt frame number 3. The attacker now knows a key stream and the corresponding IV which is as long as frame number 3.

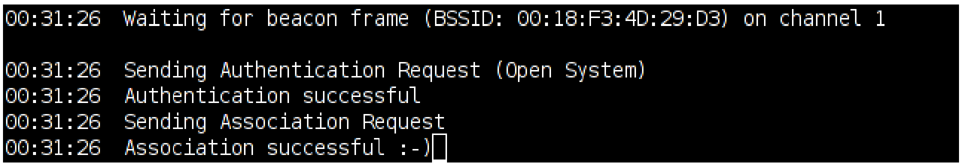
The attacker can now initiate an *Shared Key authentication* handshake with the AP. After having received frame number 2, he can construct a valid frame number 3 using his recovered key stream. The AP will be able to successfully decrypt and verify the frame and respond with *successful*. The attacker is now authenticated.

We will later see that there are some more attacks which allow key stream recovery, so that an attacker does not even need to sniff a valid handshake to recover an key stream. Additionally, there are possibilities to force a station to reauthenticate itself immediately.

5.2.1 Implementation

aircrack-ng contains an implementation of the *fake authentication attack*. To authenticate the station with the address `00:14:6C:F7:17:0E` to the access point with BSSID `00:18:F3:4D:29:D3` using the wireless interface `wifi0` and *open authentication*, an attacker has to execute the following command.

```
./aireplay-ng -1 10 -a 00:18:F3:4D:29:D3 \\
-h 00:14:6C:F7:17:0E wifi0
```



```
00:31:26  Waiting for beacon frame (BSSID: 00:18:F3:4D:29:D3) on channel 1

00:31:26  Sending Authentication Request (Open System)
00:31:26  Authentication successful
00:31:26  Sending Association Request
00:31:26  Association successful :-)
```

Figure 5.2: aircrack-ng 1.0 beta 1 fakeauth results

If the attack was successful, an output similar to the one in figure 5.2 will be displayed. Because the attacker specified the `-1 10` parameter, the attack will be repeated every 10 seconds.

5.3 KoreK's chopchop attack

KoreK's chopchop attack [Kor04a] is quite an remarkable attack on WEP. It can be summarized as follows:

Attack 3 *Chophop (2004): Let O_{crc} be an oracle, which takes an arbitrary encrypted packet and returns* true*, if the checksum in the encrypted packet was correct,* false *otherwise. If an attacker has a single encrypted packet of length l and access to such an oracle O_{crc}, he can decrypt the last m bytes of the packet and recover the last m bytes of the key stream used to encrypt the packet, with in average $128 \cdot m$ queries to the oracle and negligible computational effort.*

KoreK could show, that there is more than one way to use an access point as such an oracle.

5.3.1 Mathematical background

An arbitrary sequence of bytes can be interpreted as an element of $\mathbb{F}_2[X]$ by taking the bits of the sequence as coefficients of the polynomial. [1] Let P be this polynomial. P has a correct checksum, if and only if the equation:

$$P \bmod R_{CRC} \quad = \quad R_{ONE} \tag{5.1}$$

[1] CRC32 does this by inverting the first 32 bits first, to detect leading zero bytes prependet to the data. This makes no difference for the following explanation and is only important for implementations of this attack

holds, where R_{CRC} is the special CRC32 polynomial and R_{ONE} is the polynomial with all coefficients from X^0 to X^{31} being 1. Originally, the CRC32 checksum method was designed to detect transmission errors caused by line noise and similar effects. CRC32 was never designed to provide cryptographic protection of data. Why exactly this value gives the receiver a high chance to detect a random transmission error is out of the scope of this document.

$$\begin{aligned} R_{CRC} = {} & X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + \\ & X^{11} + X^{10} + X^{8} + X^{7} + X^{5} + X^{4} + X^{2} + X + 1 \end{aligned} \tag{5.2}$$

$$R_{ONE} = \sum_{i=0}^{31} X^i \tag{5.3}$$

Please note that R_{CRC} is irreducible over $\mathbb{F}_2[X]$ and $\mathbb{F}_2[X]/(R_{CRC})$ is a finite field.

We will now have a closer look at the one byte shortened version of P. We can write P as $Q \cdot X^8 + P_7$ with P_7 being all elements of P with exponents smaller than 8.

$$P = Q \cdot X^8 + P_7 \tag{5.4}$$

We would like to know how Q needs to be altered so that it has a correct checksum again. From equation 5.4 and 5.1, we know that $Q \cdot X^8 + P_7$ has a correct checksum:

$$Q \cdot X^8 = P_{ONE} + P_7 \bmod R_{CRC} \tag{5.5}$$

Because $\mathbb{F}_2[X]/(R_{CRC})$ is a finite field, X^8 is invertible with inverse:

$$\begin{aligned} \left(X^8\right)^{-1} & = X^{31} + X^{29} + X^{27} + X^{24} + X^{23} + X^{22} + X^{20} + X^{17} + \\ & X^{16} + X^{15} + X^{14} + X^{13} + X^{1}0 + X^{9} + X^{7} + X^{5} + X^{2} + X \\ & = R_{INV} \end{aligned} \tag{5.6}$$

We now know, that:

$$Q = R_{INV}(P_{ONE} + P_7) \bmod R_{CRC} \tag{5.7}$$

But to be a correct message, Q should have the value:

$$Q = P_{ONE} \bmod R_{CRC} \tag{5.8}$$

By adding $P_{COR} = P_{ONE} + R_{INV}(P_{ONE} + P_7)$ to Q, we get a new corrected message with correct CRC32 checksum. Because this addition and the RC4 stream cipher are both linear, this can be added too, to an encrypted packet.

This value only depends on known value and P_7. Because there are at most 256 possible values for P_7, an attacker can now start guessing a value, shorten the original message by one byte, add the correction and then query the oracle if his guess was correct. If he has guessed P_7 correctly, the oracle will return *true*, *false* otherwise.

In average, the attacker will need 128 query per byte. This results in $m \cdot 128$ queries in average for decrypting m bytes of ciphertext at the end of a packet.

5.3.2 An example

Let's assume that we are looking at the following plaintext:

$$\begin{aligned} P_{PLAIN} =& X^{39} + X^{34} + X^{31} + X^{26} + X^{24} + X^{23} + X^{22} + X^{21} \\ & X^{20} + X^{18} + X^{17} + X^{16} + X^{14} + X^{13} + X^{11} + X^{8} \\ & X^{7} + X^{6} + X^{3} + X^{2} + 1 \end{aligned} \tag{5.9}$$

The binary representation of P_{PLAIN} is:

1000010010000101111101110110100111001101

Or its hexadecimal representation is **84 85 F7 69 CD**.

We can verify that:

$$P_{PLAIN} = R_{ONE} \bmod R_{CRC} \tag{5.10}$$

So P_{PLAIN} has a valid checksum. We can rewrite P_{PLAIN} as:

$$\begin{aligned} P_{PLAIN} =& X^8 \cdot (X^{31} + X^{26} + X^{23} + X^{18} + X^{16} + X^{15} + \\ & X^{14} + X^{13} + X^{12} + X^{10} + X^9 + X^8 + X^6 + \\ & X^5 + X^3 + 1) + \\ & (X^7 + X^6 + X^3 + X^2 + 1) \end{aligned} \tag{5.11}$$

Here,

$$P_7 = X^7 + X^6 + X^3 + X^2 + 1 \tag{5.12}$$

which is the rightmost byte and

$$\begin{aligned} Q =& X^{31} + X^{26} + X^{23} + X^{18} + X^{16} + X^{15} + X^{14} + X^{13} + \\ & X^{12} + X^{10} + X^9 + X^8 + X^6 + X^5 + X^3 + 1 \end{aligned} \tag{5.13}$$

We can easily see that Q does not have a correct checksum, because $Q \neq R_{ONE} \bmod R_{CRC}$. We can now start to calculate the correction value $P_{COR} = P_{ONE} + R_{INV}(P_{ONE} + P_7)$

$$\begin{aligned} P_{COR} =& X^{30} + X^{29} + X^{28} + X^{27} + X^{25} + X^{24} + X^{22} + X^{21} \\ & X^{20} + X^{19} + X^{17} + X^{11} + X^7 + X^4 + X^2 + X \end{aligned} \tag{5.14}$$

By adding P_{COR} to Q, we now get a modified version of Q, which has a valid checksum again.

5.3.3 Arbaugh inductive attack

The *Arbaugh inductive attack* [Arb01] can be seen as the inverse version of the KoreK attack or the KoreK attack can be seen as the inverse version of the *Arbaugh inductive attack*. Arbaugh was the first person who demonstrated that the ICV can be used to extend the key stream used to encrypt a packet byte by byte. Based on the *Arbaugh inductive attack*, *KoreK* developed his *chopchop attack*.

In a nutshell, Arbaugh could show the following: If an attacker has recovered a single encrypted packet of length l and has access to O_{crc}, he can determine the next m bytes of the key stream used to encrypt this packet with in average $m \cdot 128$ queries to the oracle and negligible computational effort.

For a correctly encrypted packet, we know that the plaintext P fulfills the equation:

$$P = P_{ONE} \bmod R_{CRC} \tag{5.15}$$

Adding a single zero byte to the packet is equivalent to multiplying P with X^8. Let now Q be the one zero byte extended version of P

$$\begin{aligned} Q =& P \cdot X^8 && (5.16) \\ =& P_{ONE} \cdot X^8 \bmod R_{CRC} && (5.17) \end{aligned}$$

If we add $P_{ONE} \cdot X^8 + P_{ONE}$ to Q, the extend packet will have a valid CRC checksum again.

An attacker can now start to guess the next byte of the key stream P_{KS}, add $P_{ONE} \cdot X^8 + P_{ONE}$ and P_{KS} to Q and query the oracle, if the guess was correct. Because there are at most 256 different values for P_{KS}, the attacker will succeed after 128 guesses in average.

5.3.4 Using the AP as an O_{crc} oracle

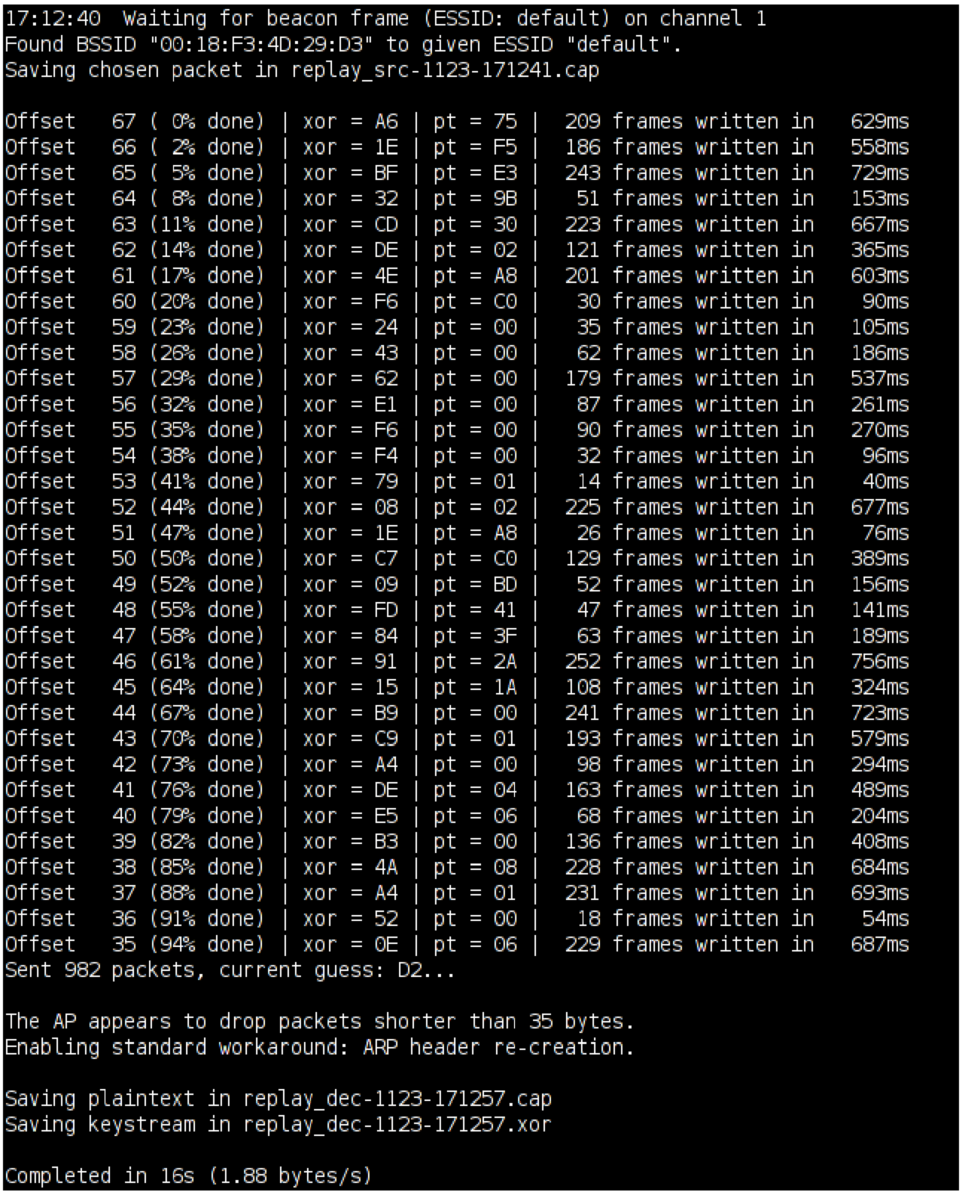
There are at least 2 ways, an attacker can use an AP as an O_{crc} oracle.

1. The attacker could join the network with two stations A and B and send the packet from station A to station B. If the checksum is correct, the packet will be relayed by the AP to station B. If the checksum was incorrect, the packet will be discarded.
2. The attacker could send the packet from a station which is not in the network to the AP. If the checksum was correct, the AP will send an error-message to the station, telling it, that it needs to rejoin the network. If the checksum was incorrect, the packet will be discarded.

5.3.5 Implementation

aircrack-ng contains an implementation of the *chopchop attack*. To execute a chopchop attack to decrypt a single packet from the *access point* with the BSSID `00:18:F3:4D:29:D3` using the client address `00:11:6B:3A:A0:26` and the wireless interface `wifi0`, an attacker has to execute the following command:

```
./aireplay-ng -4 -b 00:18:F3:4D:29:D3 -h 00:11:6B:3A:A0:26 wifi0
```



```
17:12:40  Waiting for beacon frame (ESSID: default) on channel 1
Found BSSID "00:18:F3:4D:29:D3" to given ESSID "default".
Saving chosen packet in replay_src-1123-171241.cap

Offset   67 ( 0% done) | xor = A6 | pt = 75 |  209 frames written in   629ms
Offset   66 ( 2% done) | xor = 1E | pt = F5 |  186 frames written in   558ms
Offset   65 ( 5% done) | xor = BF | pt = E3 |  243 frames written in   729ms
Offset   64 ( 8% done) | xor = 32 | pt = 9B |   51 frames written in   153ms
Offset   63 (11% done) | xor = CD | pt = 30 |  223 frames written in   667ms
Offset   62 (14% done) | xor = DE | pt = 02 |  121 frames written in   365ms
Offset   61 (17% done) | xor = 4E | pt = A8 |  201 frames written in   603ms
Offset   60 (20% done) | xor = F6 | pt = C0 |   30 frames written in    90ms
Offset   59 (23% done) | xor = 24 | pt = 00 |   35 frames written in   105ms
Offset   58 (26% done) | xor = 43 | pt = 00 |   62 frames written in   186ms
Offset   57 (29% done) | xor = 62 | pt = 00 |  179 frames written in   537ms
Offset   56 (32% done) | xor = E1 | pt = 00 |   87 frames written in   261ms
Offset   55 (35% done) | xor = F6 | pt = 00 |   90 frames written in   270ms
Offset   54 (38% done) | xor = F4 | pt = 00 |   32 frames written in    96ms
Offset   53 (41% done) | xor = 79 | pt = 01 |   14 frames written in    40ms
Offset   52 (44% done) | xor = 08 | pt = 02 |  225 frames written in   677ms
Offset   51 (47% done) | xor = 1E | pt = A8 |   26 frames written in    76ms
Offset   50 (50% done) | xor = C7 | pt = C0 |  129 frames written in   389ms
Offset   49 (52% done) | xor = 09 | pt = BD |   52 frames written in   156ms
Offset   48 (55% done) | xor = FD | pt = 41 |   47 frames written in   141ms
Offset   47 (58% done) | xor = 84 | pt = 3F |   63 frames written in   189ms
Offset   46 (61% done) | xor = 91 | pt = 2A |  252 frames written in   756ms
Offset   45 (64% done) | xor = 15 | pt = 1A |  108 frames written in   324ms
Offset   44 (67% done) | xor = B9 | pt = 00 |  241 frames written in   723ms
Offset   43 (70% done) | xor = C9 | pt = 01 |  193 frames written in   579ms
Offset   42 (73% done) | xor = A4 | pt = 00 |   98 frames written in   294ms
Offset   41 (76% done) | xor = DE | pt = 04 |  163 frames written in   489ms
Offset   40 (79% done) | xor = E5 | pt = 06 |   68 frames written in   204ms
Offset   39 (82% done) | xor = B3 | pt = 00 |  136 frames written in   408ms
Offset   38 (85% done) | xor = 4A | pt = 08 |  228 frames written in   684ms
Offset   37 (88% done) | xor = A4 | pt = 01 |  231 frames written in   693ms
Offset   36 (91% done) | xor = 52 | pt = 00 |   18 frames written in    54ms
Offset   35 (94% done) | xor = 0E | pt = 06 |  229 frames written in   687ms
Sent 982 packets, current guess: D2...

The AP appears to drop packets shorter than 35 bytes.
Enabling standard workaround: ARP header re-creation.

Saving plaintext in replay_dec-1123-171257.cap
Saving keystream in replay_dec-1123-171257.xor

Completed in 16s (1.88 bytes/s)
```

Figure 5.3: aircrack-ng 1.0 beta 1 chopchop results

If the attack was successful, output similar to the one in figure 5.3 will be displayed.

5.3.6 Implementation note

By using a lot of stations (256 for example), the attacker can send all 256 guesses in one burst and encode the guess in the last byte of the senders address. Using this method, the attacker does not need to wait until the packet has been delayed or he can be sure that the packet was discarded before sending the next packet.

This attack can be hard to detect, because packets with invalid checksums are not reported to the link- or network layer and therefore will not be seen by sniffers and IDS systems only working on the link- or network layer.

5.4 Bittau's fragmentation attack

Bittau noticed that the IEEE 802.11 protocol supports fragmentation and was able to exploit this feature [BHL06].

Attack 4 *Fragementation (2006): A client is able to split a packet into up to 16 fragments; each of them is encrypted separately. After an attacker has discovered a single key stream of length* m*, he can send packets with* $((m-4) \cdot 16) = 16 \cdot m - 64$ *bytes of arbitrary payload (length of the ICV excluded) and recover a key stream of length* $16 \cdot m - 60$ *bytes, by splitting them into up to 16 separate fragments.*

5.4.1 Technical background

After the attacker has discovered a key stream of length m, he could simply send packets with arbitrary payload of length $m-4$ (length of the 4 byte ICV excluded). To send longer packets, the attacker can split his payload into up to 16 packets of length $m-4$ bytes payload per packet. These packets are then encrypted using the discovered key stream. All packets are now marked to be fragments of a single packet. After the AP has received all fragments, the *original* packet is reassembled and, depending on the destination address, reencrypted with a new key stream and relayed by the AP as a single fragment.

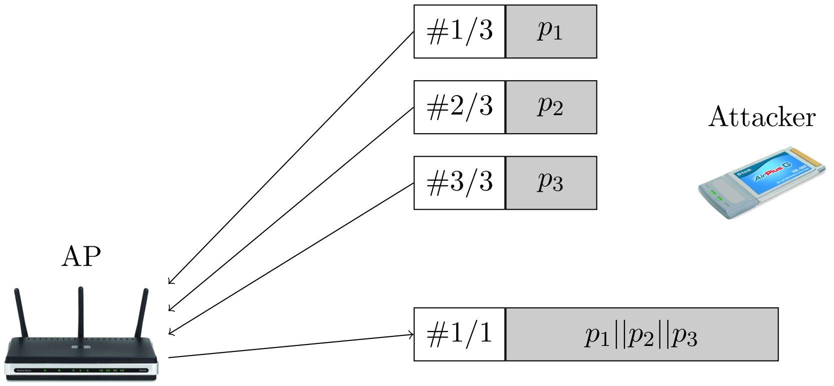


Figure 5.4: Fragmentation attack example with 3 fragments

Because the attacker knows the plaintext of the relayed packet, he can recover the key stream of the relayed packet. Now the attacker knows a key stream of length $(m-4) \cdot 16 + 4 = 16 \cdot m - 60$ bytes (he has chosen $16 \cdot m - 64$ bytes of plaintext and can calculate the 4 bytes ICV value).

Usually, an attacker can at least guess the first 8 bytes of an arbitrary encrypted Ethernet frame or sometimes even more (see Section 8.4). After having sent 16 packets, the attacker now knows a key stream of 68 bytes. After having sent 16 packets again, he knows a key stream of 1024 bytes. After having sent 2 packets, he knows a key stream of length 1504 bytes, which is the maximum size of an Ethernet packet (with ICV). [2] The attacker can now send packets with arbitrary length and payload after having received 4 and send 34 packets.

5.4.2 Advanced attack methods using fragmentation

Bittau found more than one way to exploit WEP using fragmentation. A very interesting attack is the *internet redirection attack*, which allows an attacker to redirect arbitrary intercepted packets to a host on the internet of the attacker's choice, if the AP is connected to the internet. If the attacker controls the destination host, he can intercept these packets there and just read the plaintext. All packets are decrypted by the AP before they are send to the internet.

Alternatively, an attacker who has a key stream, which is too short to send a *shared key authentication* response, can perform a *shared key authentication* by sending the response in multiple short fragments. He can also decrypt packets in a *chopchp-like* manner, starting from the beginning of the packet, instead of the last byte.

The exact technical details are described in Bittau's paper *The Final Nail in WEP's Coffin* [BHL06], which contains even more attacks using fragmentation.

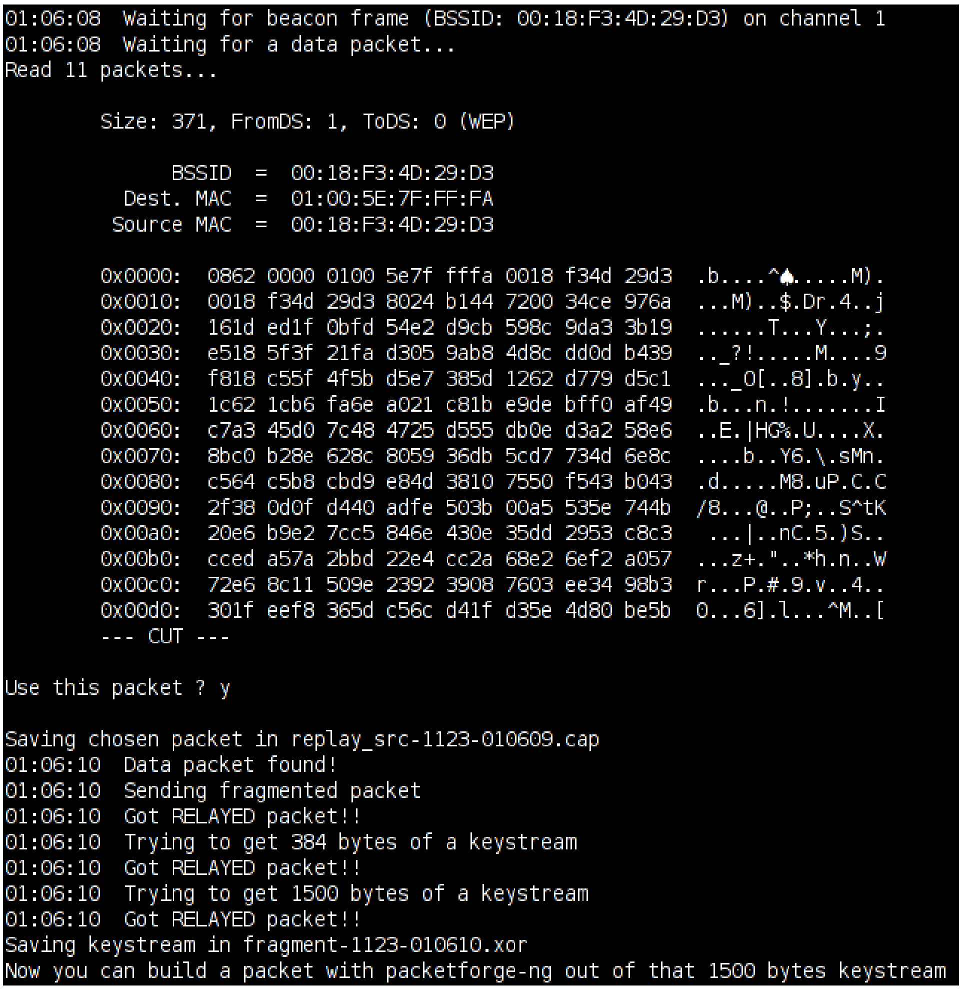
5.4.3 Implementation

aircrack-ng contains an implementation of the *fragmentation attack*. To get a key stream of maximum length from the access point with the BSSID `00:18:F3:4D:29:D3` using the wireless interface `wifi0`, an attacker has to execute the following command:

```
./aireplay-ng -5 -b 00:18:F3:4D:29:D3 wifi0
```

aircrack-ng will wait for a suitable packet and ask the attacker if it should be used, if a suitable packet was found. If the attack was successful, an output

[2] IEEE 802.11 allows up to 2304 bytes of payload (ICV excluded), but most implementations just use up to 1500 bytes.



```
01:06:08  Waiting for beacon frame (BSSID: 00:18:F3:4D:29:D3) on channel 1
01:06:08  Waiting for a data packet...
Read 11 packets...

        Size: 371, FromDS: 1, ToDS: 0 (WEP)

              BSSID  =  00:18:F3:4D:29:D3
          Dest. MAC  =  01:00:5E:7F:FF:FA
         Source MAC  =  00:18:F3:4D:29:D3

        0x0000:  0862 0000 0100 5e7f fffa 0018 f34d 29d3  .b....^♠.....M).
        0x0010:  0018 f34d 29d3 8024 b144 7200 34ce 976a  ...M)..$.Dr.4..j
        0x0020:  161d ed1f 0bfd 54e2 d9cb 598c 9da3 3b19  ......T...Y...;.
        0x0030:  e518 5f3f 21fa d305 9ab8 4d8c dd0d b439  .._?!.....M....9
        0x0040:  f818 c55f 4f5b d5e7 385d 1262 d779 d5c1  ..._O[..8].b.y..
        0x0050:  1c62 1cb6 fa6e a021 c81b e9de bff0 af49  .b...n.!.......I
        0x0060:  c7a3 45d0 7c48 4725 d555 db0e d3a2 58e6  ..E.|HG%.U....X.
        0x0070:  8bc0 b28e 628c 8059 36db 5cd7 734d 6e8c  ....b..Y6.\.sMn.
        0x0080:  c564 c5b8 cbd9 e84d 3810 7550 f543 b043  .d.....M8.uP.C.C
        0x0090:  2f38 0d0f d440 adfe 503b 00a5 535e 744b  /8...@..P;..S^tK
        0x00a0:  20e6 b9e2 7cc5 846e 430e 35dd 2953 c8c3   ...|..nC.5.)S..
        0x00b0:  cced a57a 2bbd 22e4 cc2a 68e2 6ef2 a057  ...z+."..*h.n..W
        0x00c0:  72e6 8c11 509e 2392 3908 7603 ee34 98b3  r...P.#.9.v..4..
        0x00d0:  301f eef8 365d c56c d41f d35e 4d80 be5b  0...6].l...^M..[
        --- CUT ---

Use this packet ? y

Saving chosen packet in replay_src-1123-010609.cap
01:06:10  Data packet found!
01:06:10  Sending fragmented packet
01:06:10  Got RELAYED packet!!
01:06:10  Trying to get 384 bytes of a keystream
01:06:10  Got RELAYED packet!!
01:06:10  Trying to get 1500 bytes of a keystream
01:06:10  Got RELAYED packet!!
Saving keystream in fragment-1123-010610.xor
Now you can build a packet with packetforge-ng out of that 1500 bytes keystream
```

Figure 5.5: aircrack-ng 1.0 beta 1 fragmentation results

similar to the one in figure 5.5 will be displayed. In this case, the key stream was saved in the file `fragment-1123-010610.xor` in the current directory.

6 Previous attacks on WEP related to RC4

RC4 has been a subject of extensive research in the past, and a lot of attacks on RC4 have been found. This includes distinguisher and key recovery attacks in various modes of operations of RC4. In this Section, we will only focus on key recovery attacks, which can be mounted in a WEP environment.

First, a theoretical model for a WEP network is needed. In the real world, an attacker can always passively listen to the communication and seen the whole IV of all packets, and depending on various aspects, he can guess some parts of the plaintext and therefore recover some parts of the key stream. A Oracle called O_{WEP} will be used as a model for a WEP network. Because all of the following attacks can be generalized and modified for WEP-like scenarios, a more generic description will be used. O_{WEP} has three parameters:

The parameter l_{iv} is the length of the initialization vector. We always assume that the initialization vector is repented to the main key. Modification of the following attacks for modes of operations, where the initialization vector is repented to the main key, is sometimes possible, but will not be covered by this document. For WEP, l_{iv} is always 3. Some modified versions of WEP have been discussed with a larger value for l_{iv}, but have never been standardized.

The parameter l_{hs} is the length of the secret main key. The official IEEE 802.11 standard only allows $l_{\text{hs}} = 5$, which is known as 40 or 64 Bit WEP, or $l_{\text{hs}} = 13$, which is known as 104 or 128 Bit WEP. Some vendors have implemented WEP with larger key lengths.

The parameterl_{ks} is the number of key stream bytes the oracle will return. In a WEP scenario, an attacker can guess at least the first 2 bytes of the plaintext, and therefore the first 2 bytes of the key stream, by just passively listen for packets. Using active attacks like *fragmentation* (Section 5.4) or *chopchop* (Section 5.3), an attacker can recover up to the first 1504 bytes (1500 bytes for the maximum length of an Ethernet packet + 4 byte ICV) of the key stream. In Section 8.4, we will see that an attacker can sometimes recover more bytes of the key stream, by just passively listen to traffic.

Additionally, all the following attacks would also be possible if the *generalized randomized RC4 stream cipher* would be used with an $n \neq 256$. No implementation of RC4 with $n \neq 256$ has ever been used for WEP, and for all values of n, which are not a two-pow, it is somehow unclear how the key stream should be combined with the cleartext. Therefore, I will give a generic description of all attacks, but focus on $n = 256$ for estimates how effective these attacks are. If no information about n is given for any estimate, n has the value 256.

The IEEE 802.11 standard does not specify how a station should choose a value for IV. There are three different methods which are used by most vendors. Two of them are introduced here. The third method was invented after the first key recovery attack was published in 2001 to prevent this attack. This method is introduced in Section 6.1.

Random IVs In this mode, a station chooses every IV randomly from $\{0, \ldots, n-1\}^{l_{iv}}$ independent of previous and future values from a uniform distribution. We will use the oracle O_{WEP} to simulate this method for choosing values for IV.

Counter IVs In this mode, every station keeps track of the last IV used and interprets it as an unsigned integer. When the next IV is needed, 1 is added to the last value and the result is used as IV. This has the advantage that it will take $n^{l_{iv}}$ packets before a value for IV is reused. If an attacker is able to capture two different packets $(p_1 \oplus c_1)$ and $(p_2 \oplus c_2)$, encrypted using the same IV, the value $(p_1 \oplus c_1) \oplus (p_2 \oplus c_2)$ will show the difference between the two plaintexts of the packets $p_1 \oplus p_2$. We will use the oracle O_{WEP_CTR} to simulate this method for generating values for IV.

```
Oracle $O_{WEP}(l_{iv}, l_{key}, l_{ks})$
    Rk $\leftarrow_R \{0, \ldots, n-1\}^{l_{key}}$

    while query()
        IV $\leftarrow_R \{0, \ldots, n-1\}^{l_{iv}}$
        X $\leftarrow RC4(\mathsf{IV}||\mathsf{Rk}, l_{ks})$
        output(IV, X)
```

```
Oracle $O_{WEP\_CTR}(l_{iv}, l_{key}, l_{ks})$
    Rk $\leftarrow_R \{0, \ldots, n-1\}^{l_{key}}$
    IV $\leftarrow_R \{0, \ldots, n-1\}^{l_{iv}}$
    while query()
        IV $\leftarrow \mathsf{IV} + 1$
        X $\leftarrow RC4(\mathsf{IV}||\mathsf{Rk}, l_{ks})$
        output(IV, X)
```

Most drivers and firmwares generate their initialization vectors like O_{WEP_CTR}. On some attacks on WEP, the mode used to generate the initialization vectors has a huge impact on the numbers of sessions needed to perform the attack. We will use O_{WEP_CTR} to compare the effectiveness of the following attacks.

6.1 The FMS attack

The *FMS attack* [FMS01] was the first key recovery attack against RC4 in WEP-like operating modes and was published by Fluhrer, Mantin, and Shamir in 2001. We can summarize the *FMS attack* as follows:

Attack 5 *Fluhrer, Mantin, Shamir (2001): An attacker, who has access to an oracle* $O_{WEP\_CTR}(3, 13, 1)$ *can recover the internal key of the oracle with a success probability of 50% with about 9,000,000 queries to the oracle and negligible computational effort.*

6.1.1 Mathematical background

For the rest of this Chapter, all additions and subtractions, except for probabilities, are done mod n. Additionally, the following description of the *FMS attack* is a modified version for the *generalized RC4 stream cipher* and some of the ideas of Stubblefield [SIR04] and KoreK [Kor04b] have been integrated. Stubblefield was the first person who implemented this attack against a real network. Fluhrer, Mantin, and Shamir published the theoretical background, but did not implement their attack.

We will assume that an attacker knows the first l words of a RC4 key and wants to attack $\mathsf{K}[l]$ for an $l \geq 2$. Additionally, the attacker knows the first word of output of the RC4-PRGA. The attacker can now simulate the first l steps of the RC4-KSA and knows S_l, j_l and the value of i. Let's assume that the following conditions are met:

1. $\mathsf{S}_l[1] < l$
2. $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]] = l$
3. $\mathsf{S}_l^{-1}[\mathsf{X}[0]] \neq 1$
4. $\mathsf{S}_l^{-1}[\mathsf{X}[0]] \neq \mathsf{S}_l[1]$

We will refer to this condition as the *resolved condition* and say, the RC4-KSA is in *resolved state* if all these conditions are met. Most papers just use conditions 1. and 2. as *resolved condition*, conditions 3. and 4. where later introduced by KoreK to improve the effectiveness of this attack. In the next step, $\mathsf{S}_l[j_{l+1}]$ will be swapped to $\mathsf{S}_{l+1}[l]$.

We will now have a look at the first word of output of the RC4-PRGA. The first word of output is always $\mathsf{S}_{n+1}[\mathsf{S}_{n+1}[1] + \mathsf{S}_{n+1}[\mathsf{S}_n[1]]]$. If neither $\mathsf{S}_l[1]$, $\mathsf{S}_l[\mathsf{S}_l[1]]$, nor $\mathsf{S}_{l+1}[l]$ did participate in any further swaps in the rest of the RC4-KSA, the output will be $\mathsf{S}_l[j_{j+1}]$ which is equal to $\mathsf{S}_l[j_l + \mathsf{K}[l] + \mathsf{S}_l[l]]$. With other words, if none of these swaps occur, the function:

$$\mathcal{F}_{fms}(\mathsf{K}[0], \ldots, \mathsf{K}[l-1], \mathsf{X}[0]) = \mathsf{S}_l^{-1}[\mathsf{X}[0]] - j_l - \mathsf{S}_l[l] \tag{6.1}$$

will take the value of $\mathsf{K}[l]$. If one of these values is swapped during the remaining RC4-KSA, $\mathcal{F}_{fms}$ will take a more or less random value.

This can be verified by observing the first steps of the RC4-PRGA. First we assume that neither $\mathsf{S}_l[1]$, $\mathsf{S}_l[\mathsf{S}_l[1]]$, nor $\mathsf{S}_{l+1}[l]$ did participate in any further swaps in the remaining RC4-KSA. In the first steps of the RC4-PRGA, i will be set to 1 and j will be set to $\mathsf{S}_n[1]$ which is $\mathsf{S}_l[1]$. If the first swap in the RC4-PRGA does not affect the sum $\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]]$ nor $\mathsf{S}[\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]]]$, the output will be $\mathsf{S}_l[j_{l+1}]$ which is equal to $\mathsf{S}_l[j_l + \mathsf{K}[l] + \mathsf{S}_l[l]]$. By solving this equation for $\mathsf{K}[l]$, you get the function $\mathcal{F}_{fms_l}$.

Of course, in the first swap of the RC4-PRGA will swap $\mathsf{S}[1]$ and $\mathsf{S}[\mathsf{S}[1]]$, but this will not affect the sum $\mathsf{S}[1]+\mathsf{S}[\mathsf{S}[1]]$. The only possibility for the first swap to affect the output of the first word would be, if $\mathsf{S}[\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]]]$ would be exchanged with another value, which can only happen, if $\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]] = 1$ or $\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]] = \mathsf{S}[1]$. We will check both cases separately.

1. $\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]] = 1$

   This case can never happen. We know that all these values did not participate in any swaps in the remaining RC4-KSA. So this is equivalent to $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]] = 1$. We already know that $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]] = l$ and $l \geq 2$.

2. $\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]] = \mathsf{S}[1]$

   This can never happen too. Again, we know that this is equivalent to $\mathsf{S}_l[1]+\mathsf{S}_l[\mathsf{S}_l[1]] = \mathsf{S}_l[1]$ By subtracting $\mathsf{S}_l[1]$ from both sides of the equation, we know that $\mathsf{S}_l[\mathsf{S}_l[1]] = 0$ must hold for this. Because $\mathsf{S}_l[1] < l$ and $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]] = l$ holds, we know that $\mathsf{S}_l[\mathsf{S}_l[1]] \geq 1$ and therefore cannot be 0.

If the output $\mathsf{X}[0]$ is $\mathsf{S}_l[1]$ or $\mathsf{S}_l[\mathsf{S}_l[1]]$, this would indicate that $\mathsf{S}_{l+1}[l]$ did take the value $\mathsf{S}_l[1]$ or $\mathsf{S}_l[\mathsf{S}_l[1]]$ which would mean that $\mathsf{S}_l[1]$ or $\mathsf{S}_l[\mathsf{S}_l[1]]$ was modified after step l of the RC4-KSA. Because we require $\mathsf{S}_l[1]$ and $\mathsf{S}_l[\mathsf{S}[1]]$ to remain unchanged after step l, we check for these conditions and do not use the session if condition 3. or 4. are met.

What remains is checking, with which probability none of these three values is swapped in the remaining RC4-KSA, we cannot observe. $\mathsf{S}[k]$ will only be swapped if either i or j takes the value k. i will only take values from $l + 1$ to $n - 1$. Because $l \geq 2$, i will never again take the value 1, so $\mathsf{S}_l[1]$, $\mathsf{S}_l[\mathsf{S}_l[1]]$, and $\mathsf{S}_{l+1}[\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]]]$ will not be swapped by i in the remaining RC4-KSA, because $\mathsf{S}[1] \leq l$ and $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]] = l$.

The only possibility that one of these values will be swapped in the remaining $n-l$ RC4-KSA steps is, that j takes the value 1, $\mathsf{S}[1]$, or $\mathsf{S}_l[1]+\mathsf{S}_l[\mathsf{S}_l[1]]$. We will use the *generalized randomized RC4 stream cipher* to estimate this probability. Here, j really takes values from a uniform distribution over all n possible values. Assuming that all three values are different, the probability that j does not take one of these three values in one step is $\frac{n-3}{n}$, and the probability that j does not take one of these three values in all remaining steps is $\left(\frac{n-3}{n}\right)^{n-l}$. For $n = 256$ and $l = 3$, this approximately 5.07% and for $l = 15$ approximately 5.84% which is the case for the first and last byte, in a 104 bit WEP scenario.

If two of these three values are equal, the probability that j does not take two specific values in all remaining steps of the RC4-KSA is $\left(\frac{n-2}{n}\right)^{n-l}$, which is approximately 13.75% for $n = 256$ and $l = 3$ and 15.10% for $l = 15$. An attacker might choose to put some more trust in the output of $\mathcal{F}_{fms}$ in such a special case.

6.1.2 An example

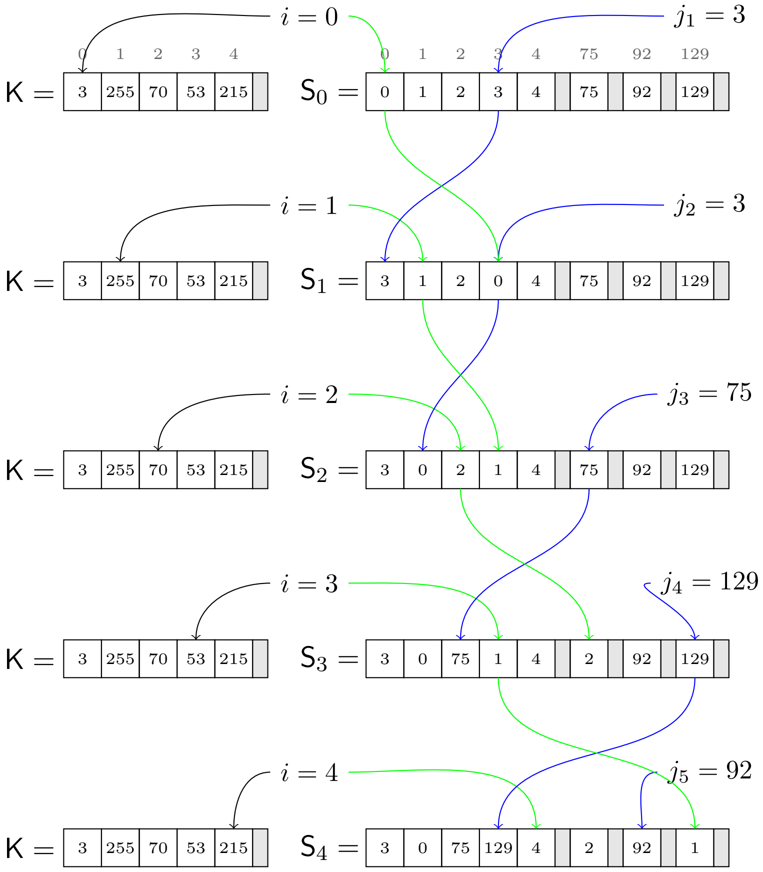

$i = 0$
$j_1 = 3$
$\mathsf{K} =$ 3 255 70 53 215
$\mathsf{S}_0 =$ 0 1 2 3 4 75 92 129
$i = 1$
$j_2 = 3$
$\mathsf{K} =$ 3 255 70 53 215
$\mathsf{S}_1 =$ 3 1 2 0 4 75 92 129
$i = 2$
$j_3 = 75$
$\mathsf{K} =$ 3 255 70 53 215
$\mathsf{S}_2 =$ 3 0 2 1 4 75 92 129
$i = 3$
$j_4 = 129$
$\mathsf{K} =$ 3 255 70 53 215
$\mathsf{S}_3 =$ 3 0 75 1 4 2 92 129
$i = 4$
$j_5 = 92$
$\mathsf{K} =$ 3 255 70 53 215
$\mathsf{S}_4 =$ 3 0 75 129 4 2 92 1

Figure 6.1: First 4 steps of RC4-KSA for $\mathsf{K} = 3, 255, 70, 53, 215, 228, 159, 214$

Let's assume that RC4 with the key $\mathsf{K} = 3, 255, 70, 53, 215, 228, 159, 214$ is used, and an attacker knows the first $l = 3$ bytes of the key K. The attacker is now trying to determine $\mathsf{K}[3]$. The attacker can compute S_3 and $j_3 = 75$ from $\mathsf{K}[0]$ to $\mathsf{K}[2]$. $\mathsf{S}_3[1] = 0$ and $\mathsf{S}_3[\mathsf{S}_3[1]] = \mathsf{S}_3[0] = 3$. Now

$$\begin{aligned} j_4 =& j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] \\ &= 75 + 1 + 53 \\ &= 129 \end{aligned} \tag{6.2}$$

and $\mathsf{S}_3[129] = 129$ is swapped to $\mathsf{S}_4[3]$. Figure 6.1 illustrates these steps in the RC4-KSA.

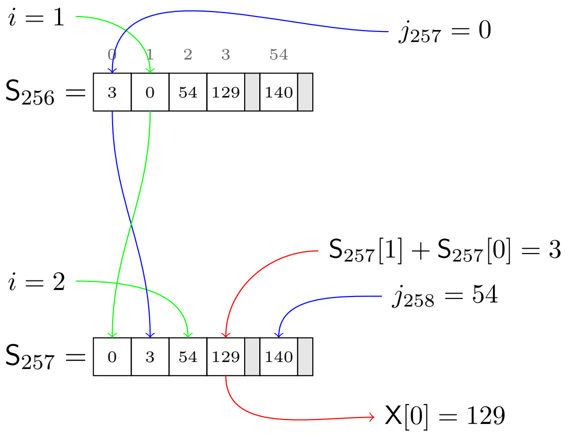


Figure 6.2: First key stream byte for $\mathsf{K} = 3, 255, 70, 53, 215, 228, 159, 214$

For the rest of the RC4-KSA, these values remain unchanged. When the first byte of output is produced by the RC4-PRGA, j_{257} is set to 0 and $\mathsf{S}_{256}[0] = 3$ and $\mathsf{S}_{256}[1] = 0$ are swapped. Now the first byte of output $\mathsf{X}[0] = \mathsf{S}_{257}[\mathsf{S}_{257}[0] + \mathsf{S}_{257}[1]] = \mathsf{S}_{257}[0+3] = \mathsf{S}_{257}[3] = 129$ is produced. Here $\mathsf{X}[0] = j_3 + \mathsf{S}_3[3] + \mathsf{K}[3]$. An attacker who would now calculate

$$\begin{aligned} \mathcal{F}_{fms_3}(3, 255, 70, 129) &= \mathsf{S}_3^{-1}[\mathsf{X}[0]] - j_3 - \mathsf{S}_3[3] \\ &= \mathsf{S}_3^{-1}[129] - 75 - 1 = 129 - 75 - 1 \\ &= 53 \end{aligned} \tag{6.3}$$

would have successfully recovered the secret key byte $\mathsf{K}[3]$. Figure 6.2 illustrates the output of the first key stream byte.

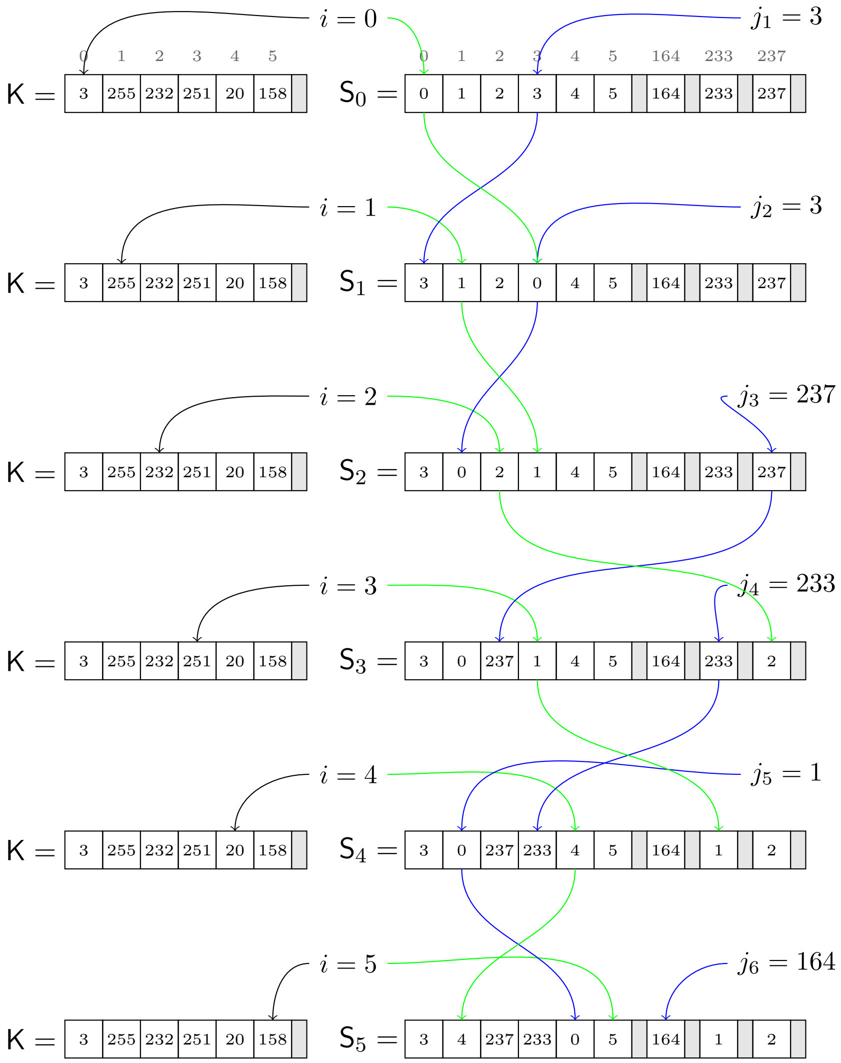


Figure 6.3: First 5 steps of RC4-KSA for $\mathsf{K} = 3, 255, 232, 251, 20, 158, 18, 173$

Let's have a look at another (unsuccessful) example. Here RC4 is used with the key $\mathsf{K} = 3, 255, 232, 251, 20, 158, 18, 173$ and the attacker again knows the first 3 bytes of the key and is interested in $\mathsf{K}[3]$. After the first 3 steps of the RC4-KSA, $j_3 = 237$, $\mathsf{S}_3[1] = 0$ and $\mathsf{S}_3[\mathsf{S}_3[1]] = 3$. Now,

$$\begin{aligned} j_4 &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] \\ &= 237 + 1 + 251 \\ &= 233 \end{aligned} \tag{6.4}$$

swaps $\mathsf{S}_3[233]$ with $\mathsf{S}_3[3] = 1$. Unfortunately, in the next step

$$\begin{aligned} j_5 &= j_4 + \mathsf{S}_4[4] + \mathsf{K}[4] \\ &= 233 + 4 + 20 \\ &= 1 \end{aligned} \tag{6.5}$$

and swaps $\mathsf{S}_4[1] = 0$ with $\mathsf{S}_4[4] = 4$. When the first byte of output by the RC4-PRGA is produced, $\mathsf{S}[1] + \mathsf{S}[\mathsf{S}[1]]$ does not point at $\mathsf{S}[3]$ anymore but at $\mathsf{S}[4]$. Therefore $\mathsf{X}[0] = 4$ and

$$\begin{aligned} \mathcal{F}_{fms_3}(3, 255, 232, 4) &= \mathsf{S}_3^{-1}[\mathsf{X}[0]] - j_3 - \mathsf{S}_3[3] \\ &= \mathsf{S}_3^{-1}[4] - 237 - 1 \\ &= 4 - 237 - 1 \\ &= 22 \\ &\neq 251 \end{aligned} \tag{6.6}$$

Figures 6.3 and 6.4 are illustrating these steps in the RC4-KSA and RC4-PRGA.

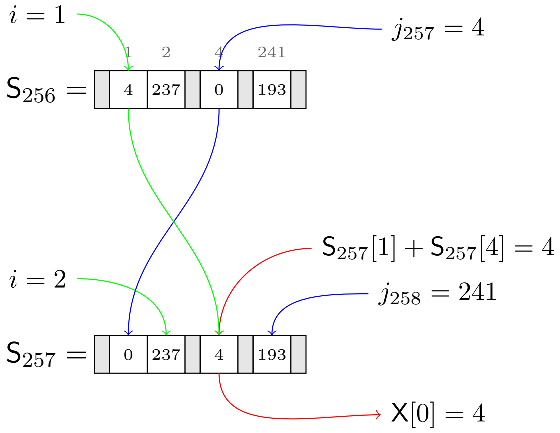


Figure 6.4: First key stream byte for $\mathsf{K} = 3, 255, 232, 251, 20, 158, 18, 173$

6.1.3 Mounting the attack

An attacker first collects some IVs and their corresponding first words of output of the RC4-PRGA using O_{WEP}. In the beginning, the attacker knows the first l words of the keys used to generate the RC4-PRGA outputs with $l = |\mathsf{IV}|$. The attacker selects a subset of these keys, where the *resolved condition* holds after the first l steps of the RC4-KSA. For all these keys, $\mathcal{F}_{fms_l}$ is computed and the most appearing result is assumed to be the next key byte $\mathsf{K}[l]$. Alternatively, one could say, that for every key, $\mathcal{F}_{fms_l}$ votes for $\mathsf{K}[l]$ having a specific value.

Now, the attacker knows the first $l+1$ words of the keys used to generate the RC4-PRGA output and can iteratively compute all remaining key bytes.

As soon as all key bytes have been computed, the resulting key can be tested for correctness, by using a few IVs and generate the corresponding key streams. If they are the same as the ones returned by the oracle, the key can be assumed to be correct with a very high probability. If not, at least one of the decisions for one of the key bytes must have been incorrect.

The attacker can now start looking for a decision for a key byte $\mathsf{Rk}[k]$, he suspects to be wrong. For example he could choose a decision where the difference in number of votes between the most voted value for $\mathsf{Rk}[k]$ and the second most voted value for $\mathsf{Rk}[k]$ is minimal. The attacker now assumes that the correct value for $\mathsf{Rk}[k]$ is the second most voted one and continue the computation with this value. This can be repeated, until the correct key has been found or a time limit has been exceeded. This is basically the same as *key ranking* [Mat94] first used by Matsui for *linear cryptanalysis*. We will discuss such methods later in Section 7.2 in detail.

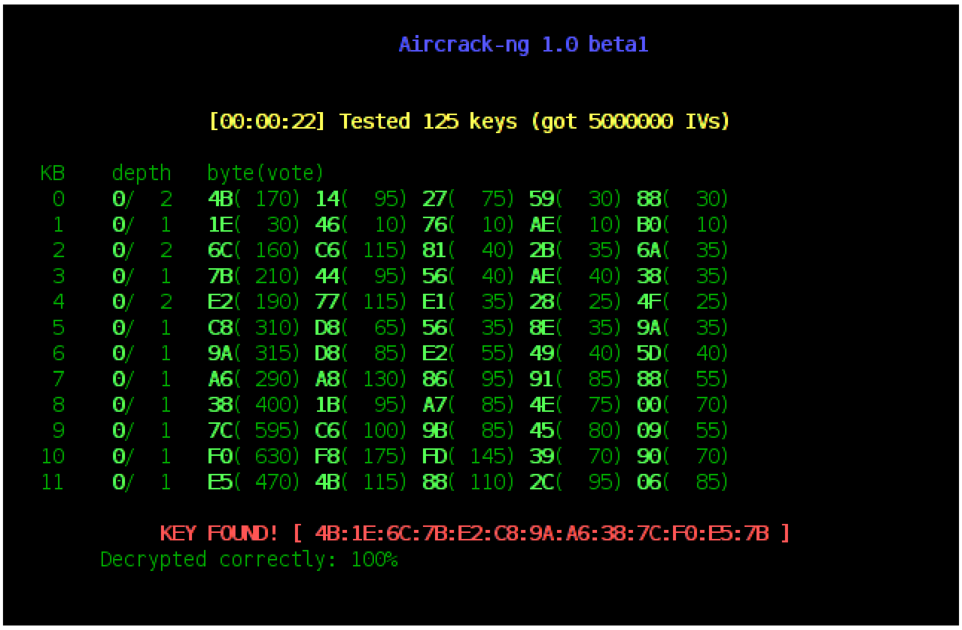
Technically, the *FMS attack* is a chosen IV attack, which means that an attacker can only use information from key streams generated using some special IVs. The condition for these initialization vectors is called *resolved condition* and the set of initialization vectors which satisfies this *resolved condition* was later called *weak initialization vectors* or *weak IVs*. Because in an WEP environment, the attacker cannot choose the initialization vector of a packet another station is going to send, he has to wait, until enough packets with these special initialization vectors have been sent.

Fluhrer, Mantin, and Shamir first suggested to use only sessions with an IV beginning with $l, 255$ to recover $\mathsf{K}[l]$. For these values for IV, it is very likely that $\mathsf{S}_l[1] < l$ and $\mathsf{S}_l[\mathsf{S}_l[1]] = l$ holds. Stubblefield suggested to simulate the first l steps of the RC4-KSA for every session and then test if these conditions are met. KoreK did the same and additionally suggested to check if $\mathsf{S}_l[1] = \mathsf{X}[0]$ or $\mathsf{S}_l[\mathsf{S}_l[1]] = \mathsf{X}[0]$, which would indicate that $\mathsf{S}_l[1]$ or $\mathsf{S}_l[\mathsf{S}_l[1]]$ has been modified in the remaining RC4-KSA.

6.1.4 Implementation

An implementation of the *FMS attack* is available in the aircrack-ng toolsuite. To start the *FMS attack* on all packets saved in the file `/tmp/fmstest.ivs`, an attacker has to execute the following command:

```
./aircrack-ng -0 -X -K -k 1 -k 2 -k 3 -k 4 -k 5 -k 7 \\
-k 8 -k 9 -k 10 -k 11 -k 12 -k 13 -k 14 -k 15 -k 16  \\
-k 17 /tmp/fmstest.ivs
```



```
                              Aircrack-ng 1.0 beta1


                 [00:00:22] Tested 125 keys (got 5000000 IVs)

   KB    depth   byte(vote)
    0    0/  2   4B( 170) 14(  95) 27(  75) 59(  30) 88(  30)
    1    0/  1   1E(  30) 46(  10) 76(  10) AE(  10) B0(  10)
    2    0/  2   6C( 160) C6( 115) 81(  40) 2B(  35) 6A(  35)
    3    0/  1   7B( 210) 44(  95) 56(  40) AE(  40) 38(  35)
    4    0/  2   E2( 190) 77( 115) E1(  35) 28(  25) 4F(  25)
    5    0/  1   C8( 310) D8(  65) 56(  35) 8E(  35) 9A(  35)
    6    0/  1   9A( 315) D8(  85) E2(  55) 49(  40) 5D(  40)
    7    0/  1   A6( 290) A8( 130) 86(  95) 91(  85) 88(  55)
    8    0/  1   38( 400) 1B(  95) A7(  85) 4E(  75) 00(  70)
    9    0/  1   7C( 595) C6( 100) 9B(  85) 45(  80) 09(  55)
   10    0/  1   F0( 630) F8( 175) FD( 145) 39(  70) 90(  70)
   11    0/  1   E5( 470) 4B( 115) 88( 110) 2C(  95) 06(  85)

               KEY FOUND! [ 4B:1E:6C:7B:E2:C8:9A:A6:38:7C:F0:E5:7B ]
        Decrypted correctly: 100%
```

Figure 6.5: aircrack-ng 1.0 beta 1 fms results

If the attack was successful, output similar to the one in figure 6.5 will be displayed. In this case, the correct key was found without doing a lot of key ranking. The correct key (except the last key byte) is displayed in the first column in the table. The numbers next to the key bytes can be seen as the number of votes for these key bytes. The numbers right to these values are the alternative candidates for the key bytes and their votes.

6.1.5 Success rate

The success rate for the *FMS attack* is quite low. If only a low number of sessions are available, the *FMS attack* works best, if all sessions are generated by O_{WEP}. If the sessions are generated by O_{WEP_CTR} or O_{WEP_LINUX}, it usually takes much more sessions. For my experimental results, I limited the CPU time available to aircrack-ng to 3 minutes. With this limit, it was only successfully less than 80% of all cases, even if the number of available sessions was very high. Because the initialization vector is only 3 bytes long in WEP, there are at most $2^{24} = 16,777,216$ possible different sessions. Other results show, if much more CPU time is available, a much higher success rate is possible.

To estimate the success rate of the *FMS attack* the aircrack-ng as described in Section 6.1.4 was used. If aircrack-ng did find the key within 3 minutes of CPU time, it was counted as a success. If aircrack-ng exited without finding the correct key or did not terminate within 3 minutes of CPU time, it was counted as failure. The total rates can be seen in figure 6.6.

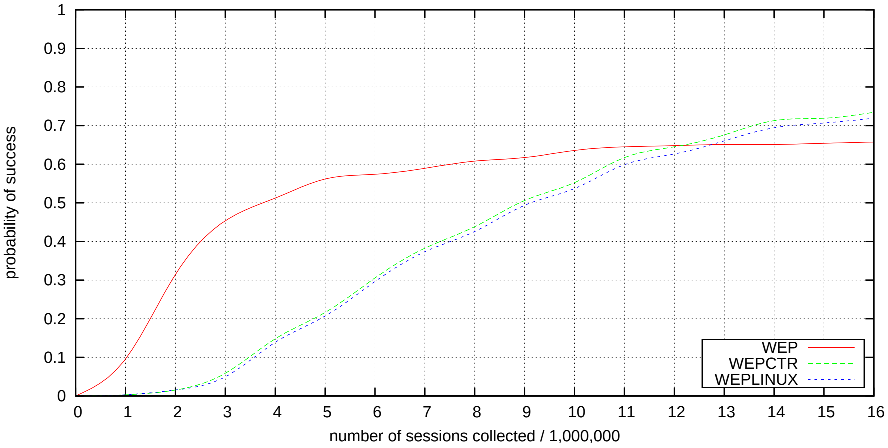


Figure 6.6: FMS success rate

6.1.6 Countermeasures

Soon after the *FMS attack* became public, developers started looking for countermeasures, without breaking compatibility with WEP. Because Fluhrer, Mantin, and Shamir suggested to use only initialization vectors starting with $3+k, 255$ to attack $\mathsf{Rk}[k]$, with $0 \leq k \leq |\mathsf{Rk}|-1$, developers started to patch their drivers and firmwares to skip these values for IV when sending packets. For example, the code in listing 6.1 is used in Linux kernel 2.6.23 to pick the initialization vector when sending a packet in a WEP protected network. Listing 6.2 shows a similar implementation for OpenBSD. Basically, this code behaves as O_{WEP_CTR} does, but skips all these *initialization vectors*. We will use the following oracle O_{WEP_LINUX} to simulate the behavior of the Linux kernel.

```
Oracle $O_{WEP\_LINUX}(l_{iv}, l_{key}, l_{ks})$
    $\mathsf{Rk} \leftarrow_R \{0,\ldots,n-1\}^{l_{key}}$
    $\mathsf{IV} \leftarrow_R \{0,\ldots,n-1\}^{l_{key}}$
    while query()
        $\mathsf{IV} \leftarrow \mathsf{IV}+1$
        if($(l_{iv} \leq \mathsf{IV}[0] \leq l_{iv}+l_{key}-1)$ && $(\mathsf{IV}[1] == n-1)$)
            $\mathsf{IV} \leftarrow \mathsf{IV}+n$
        $\mathsf{X} \leftarrow RC4(\mathsf{IV}||\mathsf{Rk}, l_{ks})$
        output$(\mathsf{IV}, \mathsf{X})$
```

Of course this does not prevent the FMS attack. It only prevents simple implementations of the FMS attack from finding the secret *root key*. Advanced implementations which simulate the first steps of the RC4-KSA and then check if the *resolved condition* holds, as described in this document in Section 6.1.1, will still be able to find enough useful sessions to determine the secret *root key*.

Listing 6.1: linux-2.6.23/net/mac80211/wep.c

```
static inline int ieee80211_wep_weak_iv(u32 iv, int keylen)
{
        /* Fluhrer, Mantin, and Shamir have reported weaknesses in the
         * key scheduling algorithm of RC4. At least IVs (KeyByte + 3,
         * 0xff, N) can be used to speedup attacks, so avoid using them. */
        if ((iv & 0xff00) == 0xff00) {
                u8 B = (iv >> 16) & 0xff;
                if (B >= 3 && B < 3 + keylen)
                        return 1;
        }
        return 0;
}


void ieee80211_wep_get_iv(struct ieee80211_local *local,
                          struct ieee80211_key *key, u8 *iv)
{
        local->wep_iv++;
        if (ieee80211_wep_weak_iv(local->wep_iv, key->keylen))
                local->wep_iv += 0x0100;

        if (!iv)
                return;

        *iv++ = (local->wep_iv >> 16) & 0xff;
        *iv++ = (local->wep_iv >> 8) & 0xff;
        *iv++ = local->wep_iv & 0xff;
        *iv++ = key->keyidx << 6;
}
```

Although this countermeasure is not a very effective one, it is still in use in a lot of wireless drivers today, and used by many open source operating systems. Lucent even started to use the marketing name *WEP Plus* for this special kind of choosing IVs.

Listing 6.2: src/sys/net80211/ieee80211_crypto.c Revision 1.36

```
        if (txflag) {
                kid = ic->ic_wep_txkey;
                wh->i_fc[1] |= IEEE80211_FC1_WEP;
                iv = ic->ic_iv ? ic->ic_iv : arc4random();
                /*
                 * Skip 'bad' IVs from Fluhrer/Mantin/Shamir:
                 * (B, 255, N) with 3 <= B < 8
                 */
                if (iv >= 0x03ff00 &&
                    (iv & 0xf8ff00) == 0x00ff00)
                        iv += 0x000100;
                ic->ic_iv = iv + 1;
```

6.2 The KoreK key recovery attack

Development did not stop after the *FMS attack* was published. Instead, people started looking for more correlations between the secret *root key* and the first bytes of output of the RC4-PRGA.

A person under the name KoreK posted [Kor04b] an implementation of a WEP cracker in 2004 in the netstumbler forum, an internet forum for users of the netstumbler tool, a famous IEEE 802.11 network scanner for windows. This implementation uses 17 different attacks, which are able to determine $\mathsf{K}[l]$, if $\mathsf{K}[0]$ to $\mathsf{K}[l-1]$ and the first two words of output $\mathsf{X}[0]$ and $\mathsf{X}[1]$ are known. Some of them had previously been known, but most of them where found by KoreK himself. KoreK assigned names like *A_u15* or *A_s13* to these attacks. For example, the original *FMS attack* has the name *A_s5_1*.

Attack 6 *KoreK (2001): An attacker, who has access to an oracle* $O_{WEP\_CTR}(3, 13, 2)$ *can recover the internal key of the oracle with success probability 50% with 700,000 queries to the oracle and negligible computational effort.*

All attacks used by KoreK can be split into three different groups:

- The first group just uses $\mathsf{K}[0]$ to $\mathsf{K}[l-1]$ and the first word of output of the RC4-PRGA $\mathsf{X}[0]$ do determine $\mathsf{K}[l]$. The original *FMS attack* is one of the attacks in this group.
- The second group additionally uses $\mathsf{X}[1]$.
- The third group is called *inverse attacks*. Instead of trying to determine the next key byte, these attacks can help to exclude certain values from being $\mathsf{K}[l]$.

I will not give a full description of all attacks used by KoreK. Instead, I will pick some of them to show how these attacks are working. Who is interested in a detailed description of those attacks can find an excellent description of them in the semester project *Break WEP Faster with Statistical Analysis* [Cha06] written by *Rafik Chaabouni*.

6.2.1 Correlation A_s13

Let's assume that the following conditions are met:

1. $\mathsf{S}_l[1] = l$
2. $\mathsf{X}[0] = l$

If $\mathsf{S}_l[1]$ did not participate in any swaps for the rest of the RC4-KSA, the following will happen in the RC4-PRGA:

1. i will be set to 1
2. j will be set to l
3. $\mathsf{S}_n[1]$ and $\mathsf{S}_n[l]$ will be swapped, now we got $\mathsf{S}_{n+1}[l] = l$ and $\mathsf{S}_{n+1}[1]$ is unknown.

If the output now is $\mathsf{X}[0] = l$, that means that $\mathsf{S}_{n+1}[l] + \mathsf{S}_{n+1}[1]$ must have pointed at $\mathsf{S}_{n+1}[l]$ and therefore $\mathsf{S}_{n+1}[1]$ and $\mathsf{S}_n[l]$ must have been 0.

If $\mathsf{S}_l[l]$ did not participate in any swaps after step $l+1$ in the RC4-KSA, j_{l+1} must have pointed at 0 in S_l, which was then swapped to $\mathsf{S}_{l+1}[l]$. This gives us the formula

$$\mathsf{S}_l^{-1}[0] = j_{l+1} = j_l + \mathsf{S}_l[l] + \mathsf{K}[l] \tag{6.7}$$

Solving this for $\mathsf{K}[l]$ gives us the following function:

$$\mathcal{F}_{korek\_A\_s13}(\mathsf{K}[0], \ldots, \mathsf{K}[l-1]) = \mathsf{S}_l^{-1}[0] - j_l - \mathsf{S}_l[l] \tag{6.8}$$

The function $\mathcal{F}_{korek\_A\_s13}$ itself does not depend on $\mathsf{X}[0]$, but $\mathsf{X}[0]$ is needed to check, if the condition $\mathsf{X}[0] = 0$ is met.

If neither $\mathsf{S}_l[1]$ nor $\mathsf{S}_{l+1}[l]$ participate in any of the remaining steps in the RC4-KSA, $\mathcal{F}_{korek\_A\_s13}$ will take the value of $\mathsf{K}[l]$. The probability of this is $\frac{n-2}{n}$ for a single step in the RC4-KSA, and $\left(\frac{n-2}{n}\right)^{n-l}$ for all remaining steps of the RC4-KSA. This is higher than the success probability of the original *FMS attack*, which needed three instead of two values in S not changing anymore in the remaining RC4-KSA. For $n = 256$ and $l = 3$, this is about 13.75%, and for $n = 256$ and $l = 15$, this is about 15.10%, which is the case for the first and last key byte in a 104 bit WEP scenario.

An example

Let's assume that RC4 with the key $\mathsf{K} = 7, 251, 14, 243, 201, 222, 52, 166$ is used and the attacker knows the first $l = 3$ bytes of the key. The attacker now tries to determine the secret key byte $\mathsf{K}[3]$.

In the next step, $j_4 = j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] = 19 + 1 + 243 = 7$ points at $\mathsf{S}_3[7] = 0$ and $\mathsf{S}_3[7] = 0$ is swapped with $\mathsf{S}_3[3] = 1$. $\mathsf{S}_3[1] = 3$ and $\mathsf{S}_4[3] = 0$ remain unchanged during the rest of the RC4-KSA. At the beginning of the RC4-PRGA, $\mathsf{S}_n[1] = 3$ and $\mathsf{S}_n[3] = 3$ are swapped and $\mathsf{S}_{n+1}[1] + \mathsf{S}_{n+1}[3] = 3$ points at $\mathsf{S}_{n+1}[3] = 3$ and the first byte of output $\mathsf{X}[0] = 3$ is produced. Figures 6.7 and 6.8 illustrate these steps.

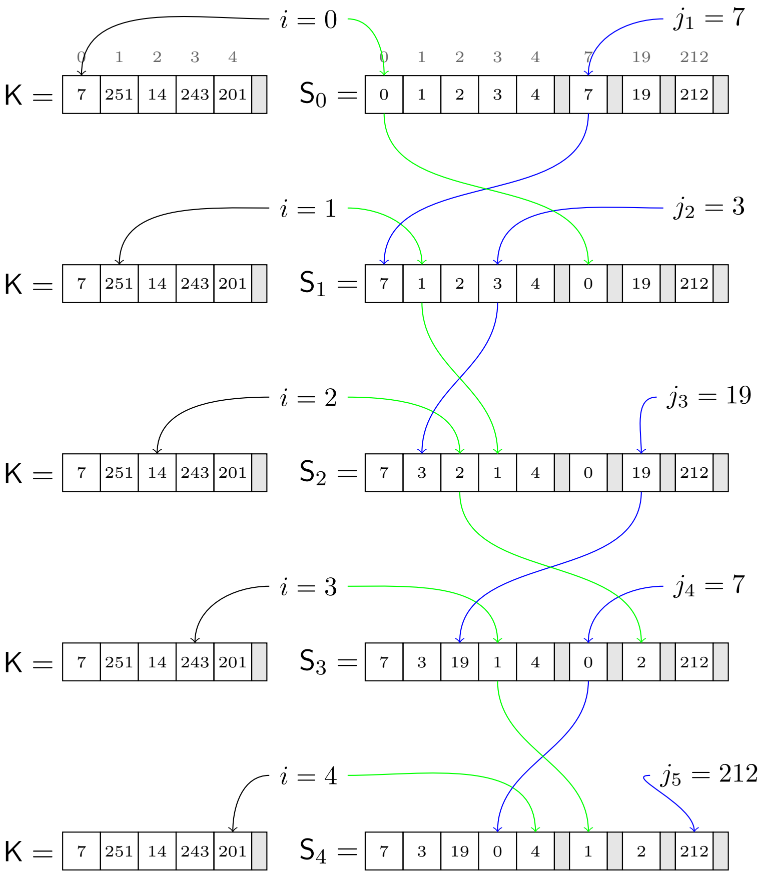


Figure 6.7: First 4 steps of RC4-KSA for $\mathsf{K} = 7, 251, 14, 243, 201, 222, 52, 166$

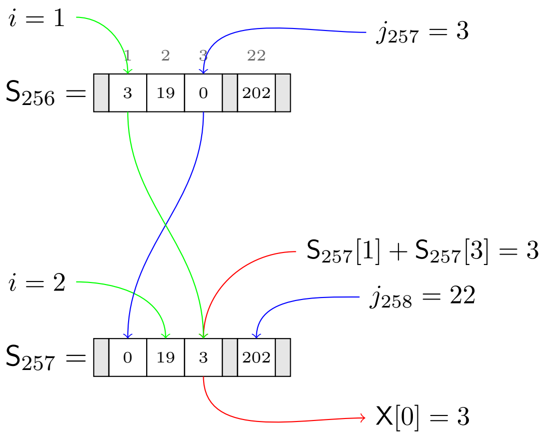


Figure 6.8: First byte of output for $\mathsf{K} = 7, 251, 14, 243, 201, 222, 52, 166$

An attacker who calculates

$$\begin{aligned} \mathcal{F}_{korek\_A\_s13}(7, 251, 14, 3) &= \mathsf{S}_3^{-1}[0] - j_3 - \mathsf{S}_3[3] \\ &= 7 - 19 - 1 \\ &= 243 \end{aligned} \tag{6.9}$$

would have gotten the correct value for $\mathsf{K}[3]$.

6.2.2 Correlation A_s3

This attack is very similar to the original *FMS attack*, but instead of $\mathsf{X}[0]$, $\mathsf{X}[1]$ is used to get information about $\mathsf{K}[l]$.

Again, we assume that the following conditions are met:

1. $\mathsf{S}_l[1] \neq 2$
2. $\mathsf{S}_l[2] \neq 0$
3. $\mathsf{S}_l[1] < l$
4. $\mathsf{S}_l[2] < l$
5. $\mathsf{S}_l[1] + \mathsf{S}_l[2] < l$
6. $\mathsf{S}_l[2] + \mathsf{S}_l[\mathsf{S}_l[1] + \mathsf{S}_l[2]] = l$
7. $\mathsf{S}_l^{-1}[\mathsf{X}[1]] \neq 1$
8. $\mathsf{S}_l^{-1}[\mathsf{X}[1]] \neq 2$
9. $\mathsf{S}_l^{-1}[\mathsf{X}[1]] \neq \mathsf{S}_l[1] + \mathsf{S}_l[2]$

As in the original *FMS attack*, $\mathsf{S}_l[j_l + \mathsf{S}_l[l] + \mathsf{K}[l]]$ is swapped to $\mathsf{S}_{l+1}[l]$. If neither $\mathsf{S}_l[1]$, $\mathsf{S}_l[2]$, nor $\mathsf{S}_{l+1}[l]$ participates in any further swaps in the remaining RC4-KSA, the following will happen in the first two steps in the RC4-PRGA:

1. i is set to 1.
2. j_{n+1} is set to $\mathsf{S}_n[1]$.
3. $\mathsf{S}_n[1]$ and $\mathsf{S}_n[\mathsf{S}_n[1]]$ are swapped. Because $\mathsf{S}_n[1] \neq 2$ and $\mathsf{S}_n[1] < l$, this will not affect $\mathsf{S}_{n+1}[2]$ or $\mathsf{S}_{n+1}[l]$. Because $\mathsf{S}_n[2] \neq 0$, this will not affect $\mathsf{S}_{n+1}[\mathsf{S}_n[1] + \mathsf{S}_n[2]]$.
4. The first word of output $\mathsf{X}[0] = \mathsf{S}_{n+1}[\mathsf{S}_{n+1}[1] + \mathsf{S}_{n+1}[\mathsf{S}_n[1]]]$ is generated. We do not use it for the attack.
5. i is set to 2.
6. j_{n+2} is set to $j_{n+1} + \mathsf{S}_{n+1}[2] = \mathsf{S}_n[1] + \mathsf{S}_n[2]$. Because $\mathsf{S}_n[1] + \mathsf{S}_n[2] < l$, $j_{n+2} < l$.
7. $\mathsf{S}_{n+1}[2]$ and $\mathsf{S}_{n+1}[\mathsf{S}_n[1] + \mathsf{S}_n[2]]$ are swapped. Because $\mathsf{S}_n[1] + \mathsf{S}_n[2] < l$, this will not affect $\mathsf{S}_{n+2}[l]$ and will not affect the sum $\mathsf{S}_{n+2}[2] + \mathsf{S}_{n+2}[\mathsf{S}_{n+2}[1] + \mathsf{S}_{n+2}[2]]$, which is still l.

8. Now, the second word of output $\mathsf{X}[1]$ is generated, which is $\mathsf{S}_{n+2}[\mathsf{S}_{n+2}[2]+\mathsf{S}_{n+2}[\mathsf{S}_{n+2}[1]+\mathsf{S}_{n+2}[2]]] = \mathsf{S}_{n+2}[l] = \mathsf{S}_l[j_l + \mathsf{S}_l[l] + \mathsf{K}[l]]$.

   If $\mathsf{S}_l^{-1}[\mathsf{X}[1]] = 1$ or $\mathsf{S}_l^{-1}[\mathsf{X}[1]] = 2$ or $\mathsf{S}_l^{-1}[\mathsf{X}[1]] = \mathsf{S}_l[1]+\mathsf{S}_l[2]$ holds, we know that $\mathsf{S}[1]$, $\mathsf{S}[2]$ or $\mathsf{S}[\mathsf{S}[1]+\mathsf{S}[2]]$ was modified in the remaining RC4-KSA.

Solving this equation for $\mathsf{K}[l]$ results in the following formula:

$$\mathcal{F}_{korek\_A\_s3}(\mathsf{K}[0], \ldots, \mathsf{K}[l-1], \mathsf{X}[1]) = \mathsf{S}_l^{-1}[\mathsf{X}[1]] - j_l - \mathsf{S}_l[l] \tag{6.10}$$

The success probability is as good as the original *FMS attack*.

This attack has initially been found by David Hulton [Hul02]. This is especially interesting, because it shows, that skipping the first word of output of the RC4-PRGA does not prevent attacks in WEP-like modes of operations.

Even if KoreK just uses the first two words of output of the RC4-PRGA, the idea behind this attack (and some other KoreK attacks) could be extended to an attack which just uses the third word of output of the RC4-PRGA. But this attack would require more values not to change for the rest of the RC4-KSA, and would have a lower success probability. For example, the probability that four different values remain unchanged by j for 252 steps in the RC4-KSA is about 2%.

An example

Let's assume that RC4 is used with the key $\mathsf{K} = 220, 255, 36, 86, 169, 80, 173, 194$. The attacker knows the first $l = 4$ bytes of K and now tries to determine $\mathsf{K}[4]$. The attacker can simulate the first 4 steps of the RC4-KSA.

In the 5th step of the RC4-KSA, $j_5 = 8$ and 8 is swapped to $\mathsf{S}_5[4]$. Additionally we got $\mathsf{S}_4[1] = 0$ and $\mathsf{S}_4[2] = 2$. Figure 6.9 illustrates these steps in the RC4-KSA.

In the first step of the RC4-PRGA, j_{n+1} is set to 0 and $\mathsf{S}_n[1]$ and $\mathsf{S}_n[0]$ are swapped. This swap does not affect $\mathsf{S}[2]$ or $\mathsf{S}[4]$. Now $\mathsf{S}_{n+1}[2] = 2$ is added to j and $j_{n+2} = 2$. The second output byte is now $\mathsf{S}_{n+2}[\mathsf{S}_{n+2}[2] + \mathsf{S}_{n+2}[2]] = \mathsf{S}_{n+2}[4] = 8$, which reveals the secret key byte $\mathsf{K}[4] = 169$ using

$$\begin{aligned} \mathcal{F}_{korek\_A\_s3}(220, 255, 36, 86, 8) &= \mathsf{S}_4^{-1}[\mathsf{X}[1]] - j_4 - \mathsf{S}_4[4] \\ &= \mathsf{S}_4^{-1}[8] - 91 - 4 \\ &= 169 \end{aligned} \tag{6.11}$$

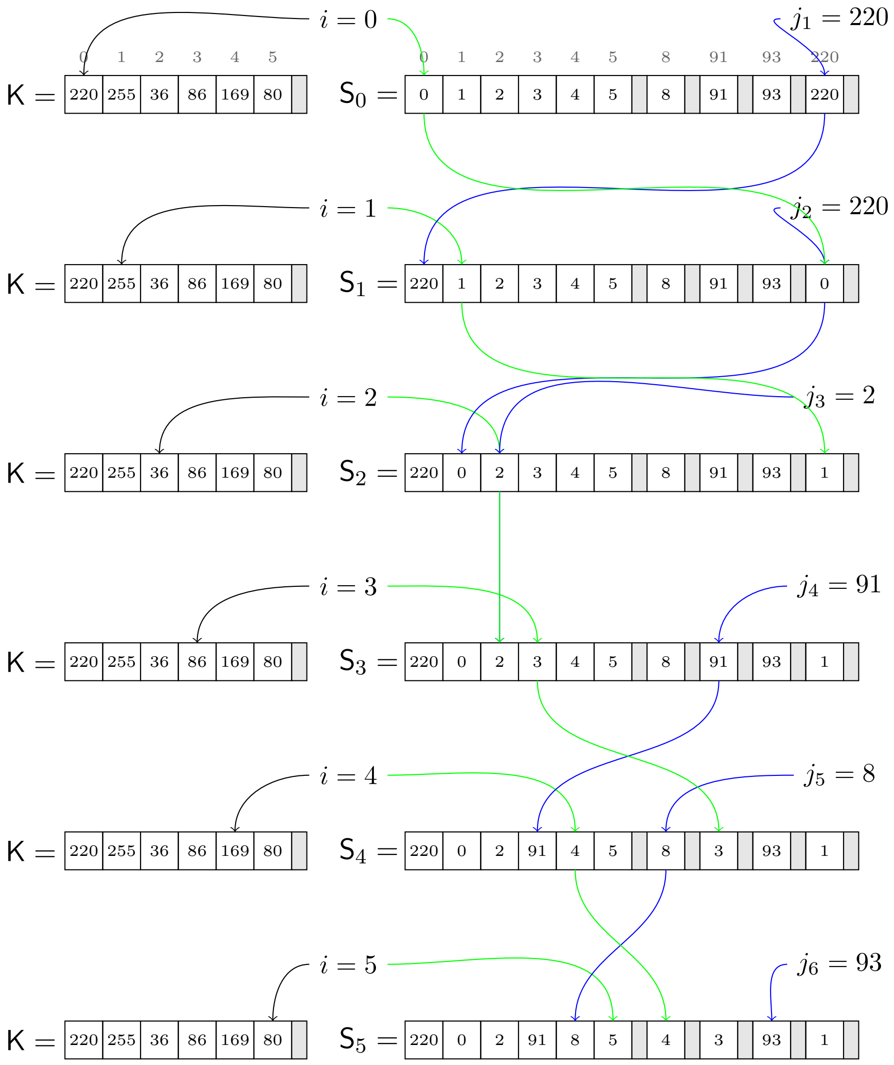

$i = 0$
$j_1 = 220$
0 1 2 3 4 5
0 1 2 3 4 5 8 91 93 220
$K =$ 220 255 36 86 169 80
$S_0 =$ 0 1 2 3 4 5 8 91 93 220
$i = 1$
$j_2 = 220$
$K =$ 220 255 36 86 169 80
$S_1 =$ 220 1 2 3 4 5 8 91 93 0
$i = 2$
$j_3 = 2$
$K =$ 220 255 36 86 169 80
$S_2 =$ 220 0 2 3 4 5 8 91 93 1
$i = 3$
$j_4 = 91$
$K =$ 220 255 36 86 169 80
$S_3 =$ 220 0 2 3 4 5 8 91 93 1
$i = 4$
$j_5 = 8$
$K =$ 220 255 36 86 169 80
$S_4 =$ 220 0 2 91 4 5 8 3 93 1
$i = 5$
$j_6 = 93$
$K =$ 220 255 36 86 169 80
$S_5 =$ 220 0 2 91 8 5 4 3 93 1

Figure 6.9: First 5 steps of RC4-KSA for $\mathsf{K} = 220, 255, 36, 86, 169, 80, 173, 194$

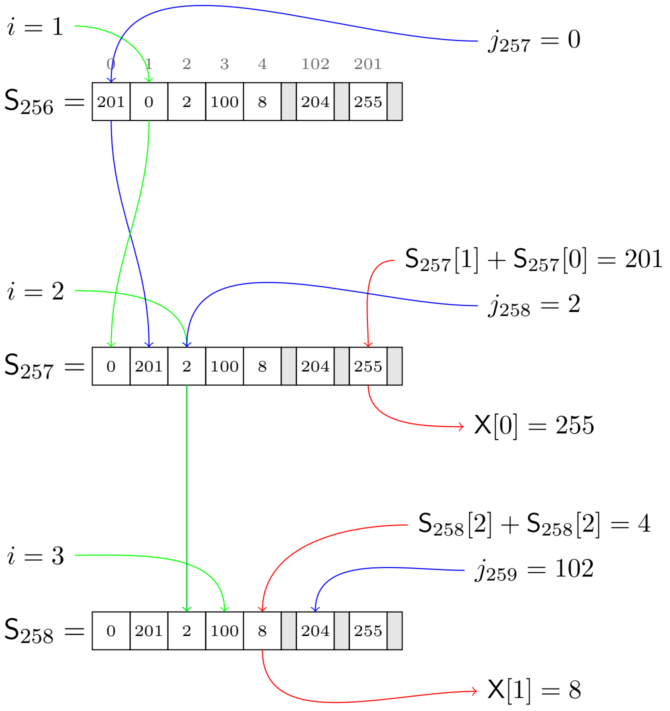


Figure 6.10: First two bytes of output for $\mathsf{K} = 220, 255, 36, 86, 169, 80, 173, 194$

6.2.3 Correlation A_neg

This attack is the most innovative one in my opinion. Let's assume that the following conditions are met:

1. $\mathsf{S}_l[1] = 2$
2. $\mathsf{S}_l[2] = 0$
3. $\mathsf{X}[0] = 2$

If $\mathsf{S}_l[1]$ and $\mathsf{S}_l[2]$ did not participate in any further swaps in the remaining RC4-KSA, the first step in the RC4-PRGA will swap $\mathsf{S}_n[1]$ and $\mathsf{S}_n[2]$ and then output $\mathsf{S}_{n+1}[2] = \mathsf{S}_n[1] = \mathsf{S}_l[1] = 2$. So if we observe this output, we can assume, that $\mathsf{K}[l]$ did not alter $\mathsf{S}_{l+1}[1]$ nor $\mathsf{S}_{l+1}[2]$. With other words, we can assume that:

1. $\mathsf{K}[l] \neq 1 - \mathsf{S}_l[l] - j_l$
2. $\mathsf{K}[l] \neq 2 - \mathsf{S}_l[l] - j_l$

Even if one of these values changes during the remaining RC4-KSA, it is highly likely, that the output will be something different than 2.

To get an estimate how good these assumptions are, we will use the following model:

- With a probability of $\left(\frac{n-2}{n}\right)^{n-l}$, these 2 values will remain unchanged in the remaining RC4-KSA, and the output will be $\mathsf{X}[0] = 2$.
- With a probability of $1 - \left(\frac{n-2}{n}\right)^{n-l}$, at least one of these values will change in the remaining RC4-KSA. In the randomized version of the RC4-PRGA, the first word of output will be 2 with a probability of $\frac{1}{n}$, and this is independent of what happens in the previous RC4-KSA.

So, the total probability that the RC4-PRGA will have 2 as the first word of output is

$$\left(\frac{n-2}{n}\right)^{n-l} + \left(1 - \left(\frac{n-2}{n}\right)^{n-l}\right) \cdot \frac{1}{n} \tag{6.12}$$

This leads to the following probability, that $\mathsf{S}[1]$ and $\mathsf{S}[2]$ remained unchanged, if the first word of RC4-PRGA output is 2:

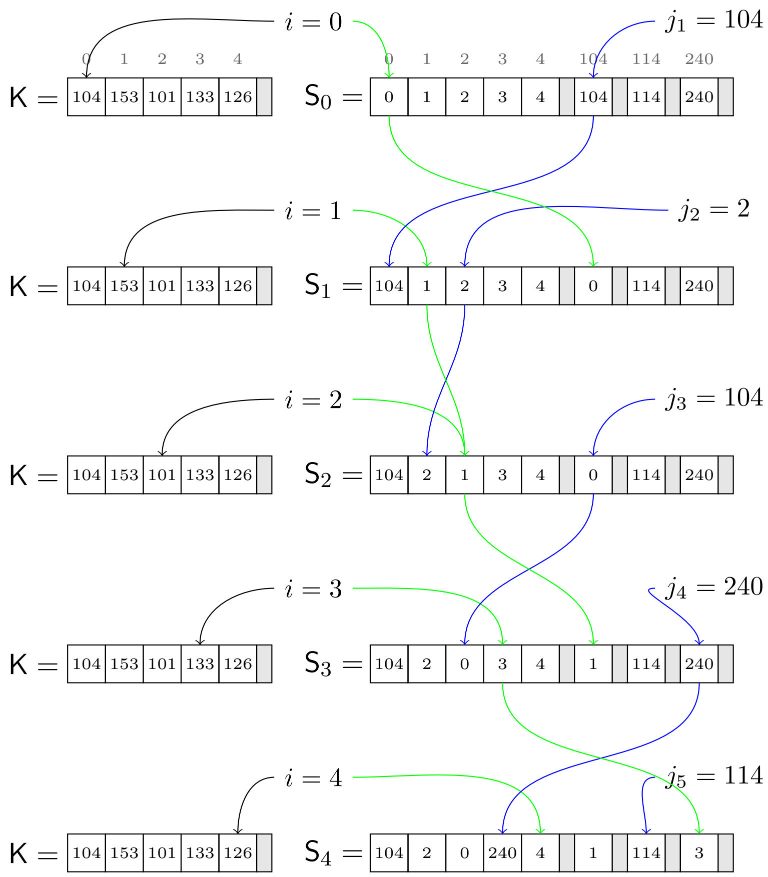
$$\frac{\left(\frac{n-2}{n}\right)^{n-l}}{\left(\frac{n-2}{n}\right)^{n-l} + \left(1 - \left(\frac{n-2}{n}\right)^{n-l}\right) \cdot \frac{1}{n}} \tag{6.13}$$

For $l = 3$ and $n = 256$, this is about 97.61% and for $l = 15$ and $n = 256$ this is 97.85%, which is the case for the first and last key byte in a 104 bit WEP scenario. This is much higher than any other success probability, we have seen before.

This is only a part of the A_neg attack. KoreK found some more criteria, which can be used to exclude certain values from being the next key byte.

An example

Let's assume that RC4 is used with the key $\mathsf{K} = 104, 153, 101, 133, 126, 174, 180, 135$ and the attacker knows the first $l = 3$ bytes of K. The attacker now tries to exclude certain values from being $\mathsf{K}[3]$. The attacker knows that $\mathsf{S}_3[1] = 2$ and $\mathsf{S}_3[2] = 0$ holds and observes $\mathsf{X}[0] = 2$.



$i = 0$
$j_1 = 104$
0 1 2 3 4
0 1 2 3 4 104 114 240
$\mathsf{K} =$ 104 153 101 133 126
$\mathsf{S}_0 =$ 0 1 2 3 4 104 114 240
$i = 1$
$j_2 = 2$
$\mathsf{K} =$ 104 153 101 133 126
$\mathsf{S}_1 =$ 104 1 2 3 4 0 114 240
$i = 2$
$j_3 = 104$
$\mathsf{K} =$ 104 153 101 133 126
$\mathsf{S}_2 =$ 104 2 1 3 4 0 114 240
$i = 3$
$j_4 = 240$
$\mathsf{K} =$ 104 153 101 133 126
$\mathsf{S}_3 =$ 104 2 0 3 4 1 114 240
$i = 4$
$j_5 = 114$
$\mathsf{K} =$ 104 153 101 133 126
$\mathsf{S}_4 =$ 104 2 0 240 4 1 114 3

Figure 6.11: First 4 steps of KSA for $\mathsf{K} = 104, 153, 101, 133, 126, 174, 180, 135$

After the 3. step of the RC4-KSA $j_3 = 104$ and i points at $\mathsf{S}_3[3] = 3$. Only $\mathsf{K}[3] = 150$ or $\mathsf{K}[3] = 151$ would have led to a change of $\mathsf{S}[1]$ or $\mathsf{S}[2]$ in the 4. step, but $\mathsf{K}[3] = 133$. For the rest of the RC4-KSA, $\mathsf{S}_3[1] = 2$ and $\mathsf{S}_3[2] = 0$ remain unchanged and the first byte of output is $\mathsf{X}[0] = 2$. The attacker can therefore exclude 150 and 151 from being $\mathsf{K}[3]$ with a high probability. Figures 6.11 and 6.12 illustrate these steps in the RC4-KSA and RC4-PRGA.

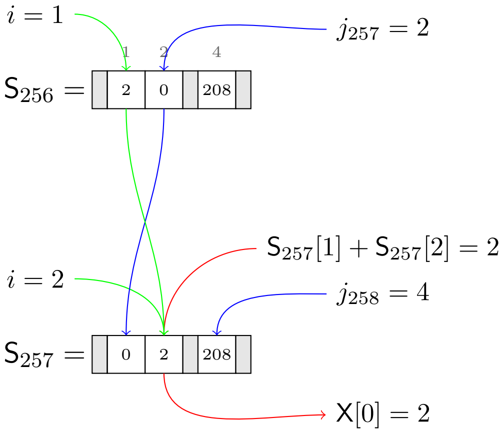


Figure 6.12: First byte of output for $\mathsf{K} = 104, 153, 101, 133, 126, 174, 180, 135$

6.2.4 Mounting the attack

As in the *FMS attack*, an attacker starts querying the oracle. Again, the attacker knows the first $l = l_{iv}$ bytes of the key K, which was used to generate the key stream. The attacker now checks for every packet, if one of the criteria of the KoreK attack is met for the known key and key stream. If so, a guess for the next key byte $\mathsf{K}[l]$ is made, and we call it a vote for $\mathsf{K}[l]$ having a specific value. The only exception is the *A_neg* attack, which votes for $\mathsf{K}[l]$ not having a specific value.

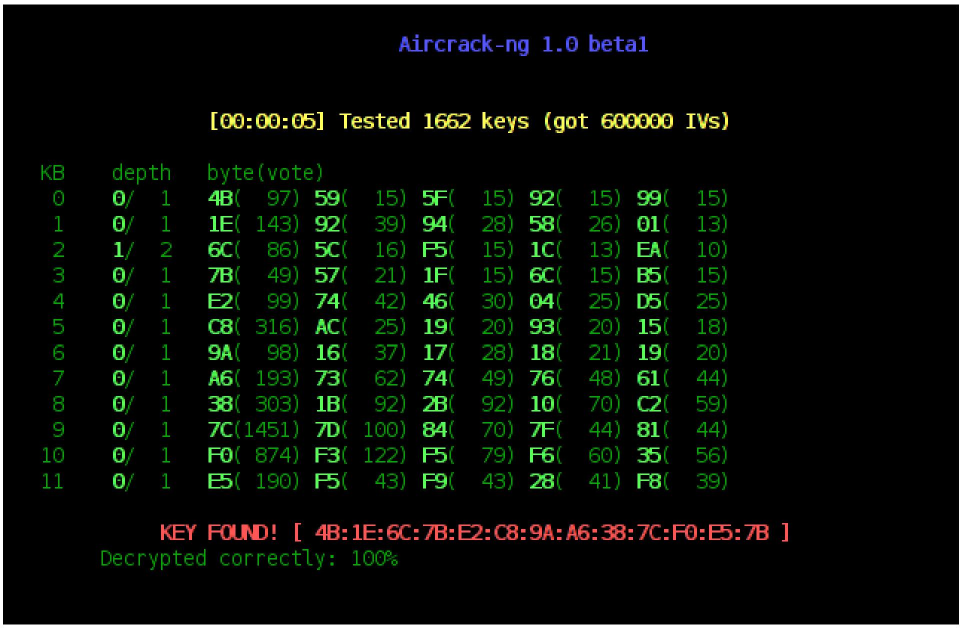
After all keys have been processed, a linear combination of all votes is calculated. The coefficients depend on the success probability of the different attacks (an attack with higher success probability has a higher coefficient than an attack with a lower success probability) and are all positive, except for *A_neg*. Now, the value with the highest number of votes in this result is assumed to be $\mathsf{K}[l]$, and the attacker knows the first $l + 1$ key bytes of all keys.

6.2.5 Implementation

As the *FMS attack*, an implementation of the *KoreK attack* is available in the *aircrack-ng* toolsuite. To execute an *KoreK attack* on all packets captured in `/tmp/korektest.ivs`, an attacker has to execute the following command:

```
./aircrack-ng -X -K /tmp/korektest.ivs
```

If the attack was successful, an output similar to the one in figure 6.13 will be displayed.



```
                                 Aircrack-ng 1.0 beta1


                    [00:00:05] Tested 1662 keys (got 600000 IVs)

   KB    depth   byte(vote)
    0    0/  1   4B(  97) 59(  15) 5F(  15) 92(  15) 99(  15)
    1    0/  1   1E( 143) 92(  39) 94(  28) 58(  26) 01(  13)
    2    1/  2   6C(  86) 5C(  16) F5(  15) 1C(  13) EA(  10)
    3    0/  1   7B(  49) 57(  21) 1F(  15) 6C(  15) B5(  15)
    4    0/  1   E2(  99) 74(  42) 46(  30) 04(  25) D5(  25)
    5    0/  1   C8( 316) AC(  25) 19(  20) 93(  20) 15(  18)
    6    0/  1   9A(  98) 16(  37) 17(  28) 18(  21) 19(  20)
    7    0/  1   A6( 193) 73(  62) 74(  49) 76(  48) 61(  44)
    8    0/  1   38( 303) 1B(  92) 2B(  92) 10(  70) C2(  59)
    9    0/  1   7C(1451) 7D( 100) 84(  70) 7F(  44) 81(  44)
   10    0/  1   F0( 874) F3( 122) F5(  79) F6(  60) 35(  56)
   11    0/  1   E5( 190) F5(  43) F9(  43) 28(  41) F8(  39)

                KEY FOUND! [ 4B:1E:6C:7B:E2:C8:9A:A6:38:7C:F0:E5:7B ]
        Decrypted correctly: 100%
```

Figure 6.13: aircrack-ng 1.0 beta 1 KoreK results

6.2.6 Success rate

The *KoreK attack* has a much better success rate than the *FMS attack*. Again, 3 minutes of CPU time was given to *aircrack-ng* to complete the attack.

The *KoreK attack* reaches a success rate of up to 97%, if a high enough amount of packets are available. The *KoreK attack* is much faster if the initialization vectors are generated randomly, instead of a sequential counter. The Linux IV generating code results in nearly no noticeable difference to a plain counter mode.

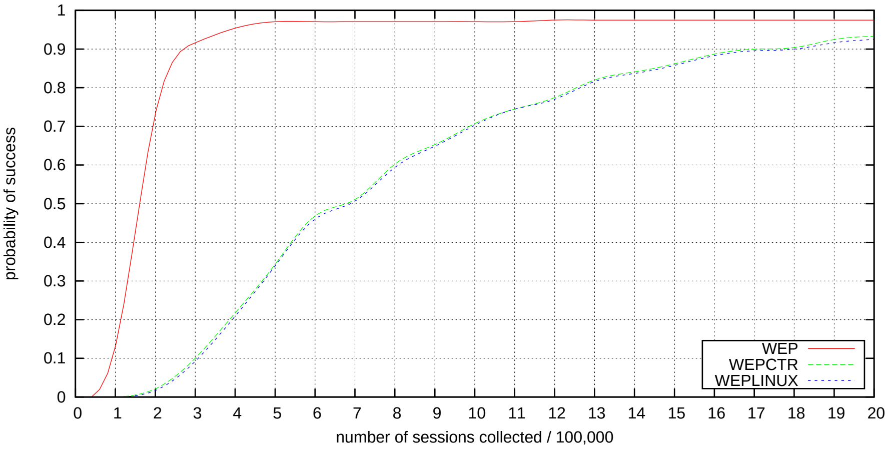


Figure 6.14: KoreK success rate

6.3 Mantin's second round attack from ASIACRYPT'05

In their final section of the paper about the *FMS attack* [FMS01], Fluhrer, Mantin, and Shamir suggested to skip the first 256 bytes of output of the RC4-PRGA. *RSA Security* made the same suggestions [Lab01] in response to the *FMS attack*:

> RSA Security has discouraged such key derivation methods, recommending instead that users consider strengthening the key scheduling algorithm by preprocessing the base key and any counter or initialization vector by passing them through a hash function such as MD5. Alternatively, weaknesses in the key scheduling algorithm can be prevented by discarding the first 256 output bytes of the pseudo-random generator before beginning encryption. Either or both of these techniques suffice to defeat the new attacks on WEP and WEP2.

So we should note, that RSA still considered RC4 to be safe, if the first 256 bytes of output are discarded, even if a simple key generation method as in WEP is used to generate the per packet key.

In 2005, *Itsik Mantin* published his paper *A Practical Attack on the Fixed RC4 in the WEP Mode* [Man05] where he showed, that RC4 in WEP-like modes can even be attacked, if the first 256 bytes of output are unavailable.

So we need a new theoretical model for this kind of attack scenario. We define a new oracle $O_{SKIPWEP}$, which uses O_{WEP} and skips the first l_{skip} bytes of its output.

```
Oracle $O_{SKIPWEP}(O_{WEP}, l_{skip})$
    while query()
        $(\mathsf{IV}, \mathsf{X}) \leftarrow \text{ask}(O_{WEP})$
        $\text{output}(\mathsf{IV}, \mathsf{X}[l_{skip}] || \ldots || \mathsf{X}[\text{length}(\mathsf{X}) - 1])$
```

Mantin showed the following:

Attack 7 *Mantin(2005): An attacker who has access to an Oracle $O_{SKIPWEP}(O_{WEP}(l_{iv}, 16, 1), 256)$ can recover the secret key of Oracle O_{WEP} with a probability of 80% by sending about $2^{25} \approx 3.3 \cdot 10^7$ queries to $O_{SKIPWEP}$ and by testing up to $2^{48} \approx 2.8 \cdot 10^{14}$ different keys for correctness, even if he has not got direct access to O_{WEP}. Different tradeoffs for CPU-time and number of queries are possible. l_{iv} must be large enough to generate 2^{25} different IVs. The complexity decreases for shorter root keys.*

This attack had no practical impact on WEP, because the first one or two bytes of the RC4-PRGA output can most times easily be recovered, and it is usually difficult to recover the 257^{th} byte of output of the RC4-PRGA. Additionally, the time and data complexity was much worse than the original FMS attack. However, Mantin could successfully show, that skipping the first 256 bytes of output of the RC4-PRGA is not sufficient to prevent key recovery attacks, if RC4 is used in a WEP-like mode.

For us, this attack is of interest, because it was the first key recovery attack on RC4, which made use of the so called *Jenkins' correlation* [Man05, MS02, Jen96, Kle06] which is also called *Glimpse property* or *RC4 Glimpse*.

6.3.1 The Jenkins' correlation

In 1996, *Robert J. Jenkins* published an observation on RC4 and its properties as a random number generator. Jenkins noted, that:

$$\text{Prob}\left(\mathsf{S}[\mathsf{S}[i]+\mathsf{S}[j]]+\mathsf{S}[j]=i\right)=\frac{2}{256} \tag{6.14}$$

holds for a random internal state of the RC4-PRGA. This is twice as high, as someone would intuitively expect it to be.

This property was later generalized and proven by different researchers [Man05, Kle06]. While *Mantin* has written a more general generalization [Man05] of the *Jenkins' correlation*, I think that *Klein* [Kle06] has found a more exact proof.

In a nutshell, Klein could show the following probabilities:

Theorem 1 *Let $n \geq 2$ and $i \in \{0,\dots,n-1\}$ be arbitrary but fixed values. For every value $x \in \{0,\dots,n-1\}$ and a randomly chosen permutation S of the numbers $0,\dots,n-1$, we got:*

$$\text{Prob}(\mathsf{S}[j]+\mathsf{S}[\mathsf{S}[i]+\mathsf{S}[j]]=i \mid j=x)=\frac{2}{n} \tag{6.15}$$

and for every value $c \in \{0,\dots,n-1\}$ with $c \neq i$:

$$\text{Prob}(\mathsf{S}[j]+\mathsf{S}[\mathsf{S}[i]+\mathsf{S}[j]]=c \mid j=x)=\frac{n-2}{n(n-1)} \tag{6.16}$$

From that follows that independently of the value of j, we got the following probability for every value of $n \geq 2$ and $i \in \{0,\dots,n-1\}$ and a randomly chosen permutation of the numbers $\{0,\dots,n-1\}$:

$$\mathrm{Prob}(\mathsf{S}[j] + \mathsf{S}[\mathsf{S}[i] + \mathsf{S}[j]] = i) = \frac{2}{n} \tag{6.17}$$

and for every value $c \in \{0, \ldots, n-1\}$ *with* $c \neq i$*:*

$$\mathrm{Prob}(\mathsf{S}[j] + \mathsf{S}[\mathsf{S}[i] + \mathsf{S}[j]] = c) = \frac{n-2}{n(n-1)} \tag{6.18}$$

This can be used to gather information about the internal state of the RC4 stream cipher. Let's assume we observe an output

$$\mathsf{X}[l] = \mathsf{S}_{n+l+1}[\mathsf{S}_{n+l+1}[l+1] + \mathsf{S}_{n+l+1}[j_{n+l+1}]] \tag{6.19}$$

We now know that:

$$\begin{aligned} \mathrm{Prob}(\mathsf{S}_{n+l+1}[j_{n+l+1}] + \mathsf{X}[l] &= l+1) = \frac{2}{n} && (6.20) \\ \mathrm{Prob}(\mathsf{S}_{n+l+1}[j_{n+l+1}] &= l+1-\mathsf{X}[l]) = \frac{2}{n} && (6.21) \end{aligned}$$

And for every $c \neq \mathsf{S}_{n+l+1}[j_{n+l+1}]$:

$$\mathrm{Prob}(c = l+1-\mathsf{X}[l]) = \frac{n-2}{n(n-1)} \tag{6.22}$$

And we know that $\mathsf{S}_{n+l+1}[j_{n+l+1}]$ was just swapped with $\mathsf{S}_{n+l}[l+1]$. So we can rewrite these equations to:

$$\begin{aligned} \mathrm{Prob}(\mathsf{S}_{n+l}[l+1] &= l+1-\mathsf{X}[l]) = \frac{2}{n} && (6.23) \\ \mathrm{Prob}(c &= l+1-\mathsf{X}[l]) = \frac{n-2}{n(n-1)} && (6.24) \end{aligned}$$

6.3.2 Mathematical background

This correlation can be used to recover the secret key of $O_{SKIPWEP}$. First let's assume that an attacker knows $\mathsf{K}[0] \ldots \mathsf{K}[l-1]$ and he wants to recover $\mathsf{K}[l]$. The attacker can now simulate the first l steps of the RC4-KSA. Let's assume that $\mathsf{S}_l[1] = l$ holds. In the next step, a value $k = \mathsf{S}_l[j_l + \mathsf{S}_l[l] + \mathsf{K}[l]]$ is swapped to $\mathsf{S}_{l+1}[l]$. Knowledge of k would reveal $\mathsf{K}[l]$, as it does in the original *FMS attack*.

With a probability of about $\left(\frac{n-2}{n}\right)^{n-l-1}$, $\mathsf{S}[1]$ and $\mathsf{S}[l]$ will remain unchanged in the remaining KSA. In the first step of the RC4-PRGA, i will be set to 1 and j will be set to $\mathsf{S}_n[1] = l$. Now, $\mathsf{S}_n[1]$ and $\mathsf{S}_n[l]$ will be swapped and $\mathsf{S}_{n+1}[1]$ takes the value k. The actual output produced by the RC4-PRGA cannot be observed, because the first n words of output of the RC4-PRGA are skipped by $O_{SKIPWEP}$. With a probability of about $\left(\frac{n-1}{n}\right)^{n-1}$, this value will remain unchanged in the next $n-1$ steps of the RC4-PRGA. We do not care about $\mathsf{S}[l]$ anymore after the first step of the RC4-PRGA.

We now have a look at step $n+1$ of the RC4-PRGA where the first output byte we can observe is produced. $\mathsf{S}_{n+n+1}[1]$ still contains the interesting value k. Using *Jerkins' correlation*, we know that $1 - \mathsf{X}[n+1]$ will be $\mathsf{S}_{n+n+1}[1]$ with a probability of $\frac{2}{n}$. If $\mathsf{S}_{n+n+1}[1] \neq k$ holds, $1 - \mathsf{X}[n+1] = k$ holds with a probability of $\frac{n-2}{n(n-1)}$. In total, we can say that

$$\text{Prob}(1 - \mathsf{X}[n+1] = k) \approx q\left(\frac{2}{n}\right) + (1-q)\left(\frac{n-2}{n(n-1)}\right) \tag{6.25}$$

$$q = \left(\frac{n-2}{n}\right)^{n-l-1}\left(\frac{n-1}{n}\right)^{n-1} \tag{6.26}$$

If $1 - \mathsf{X}[n+1] = k = \mathsf{S}_l[j_l + \mathsf{S}_l[l] + \mathsf{K}[l]]$ holds,

$$\mathcal{F}_{Mantin}(\mathsf{K}[0], \ldots, \mathsf{K}[l-1], \mathsf{X}[n+1]) = \mathsf{S}_l^{-1}[1 - \mathsf{X}[n+1]] - j_l - \mathsf{S}_l[l] \tag{6.27}$$

will take the value of $\mathsf{K}[l]$.

The actual probability for $\text{Prob}(1 - \mathsf{X}[n+1] = k)$ might even be a little bit higher, because $\mathsf{S}_{n+n+1}[1] = k$ could hold by chance, even if $\mathsf{S}[1]$ or $\mathsf{S}[l]$ where modified in the remaining RC4-KSA or $\mathsf{S}[1]$ in step 2 to n in the RC4-PRGA.

Mantin used a slightly different theoretical model for estimating $\text{Prob}(1-\mathsf{X}[n+1] = k)$, which gives a slightly higher probability for $\text{Prob}(1 - \mathsf{X}[n+1] = k)$. In our model, the probability for $n = 256$ and $l = 3$, the probability for $\text{Prob}(\mathsf{X}[n+1] = k)$ is approximately $\frac{1.0474}{256}$. Mantin did empirical tests and estimated, that $\text{Prob}(1 - \mathsf{X}[n+1] = k)$ is approximately $\frac{1.075}{256}$ for $n = 256$ and a not further specified value for l. (But we can assume that l was reasonably small for these tests)

6.3.3 An example

Let's assume that RC4 with the key $\mathsf{K} = 221, 37, 135, 232, 150, 200, 133, 253$ is used. The attacker knows the first $l = 3$ bytes of the key and is interested in $\mathsf{K}[3]$.

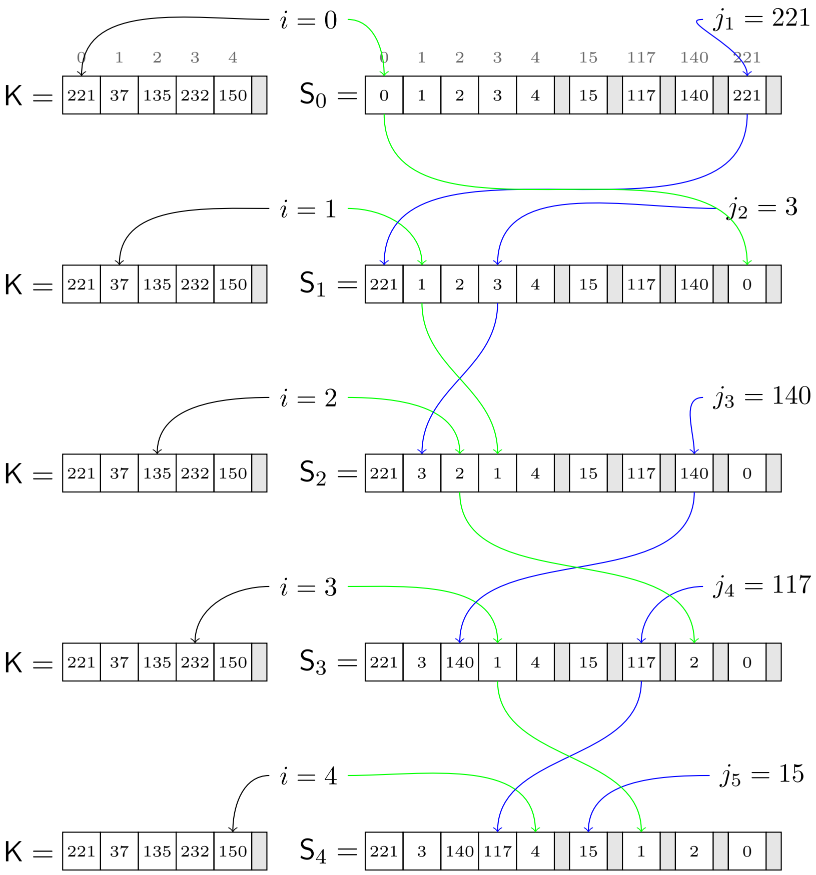

$i = 0$
$j_1 = 221$
0 1 2 3 4
0 1 2 3 4 15 117 140 221
$\mathsf{K} =$ 221 37 135 232 150
$\mathsf{S}_0 =$ 0 1 2 3 4 15 117 140 221
$i = 1$
$j_2 = 3$
$\mathsf{K} =$ 221 37 135 232 150
$\mathsf{S}_1 =$ 221 1 2 3 4 15 117 140 0
$i = 2$
$j_3 = 140$
$\mathsf{K} =$ 221 37 135 232 150
$\mathsf{S}_2 =$ 221 3 2 1 4 15 117 140 0
$i = 3$
$j_4 = 117$
$\mathsf{K} =$ 221 37 135 232 150
$\mathsf{S}_3 =$ 221 3 140 1 4 15 117 2 0
$i = 4$
$j_5 = 15$
$\mathsf{K} =$ 221 37 135 232 150
$\mathsf{S}_4 =$ 221 3 140 117 4 15 1 2 0

Figure 6.15: First steps of KSA for $\mathsf{K} = 221, 37, 135, 232, 150, 200, 133, 253$

The attacker can simulate the first 3 steps of the RC4-KSA. In the next step, $j_4 = 117$ and $\mathsf{S}_3[j_4] = 117$ is swapped with $\mathsf{S}_4[3]$. Knowledge of $\mathsf{S}_4[3]$ would reveal j_4 and $\mathsf{K}[3]$. Figure 6.15 contains a illustration of these steps.

$\mathsf{S}_4[3]$ and $\mathsf{S}_4[1]$ remain unchanged for the rest of the RC4-KSA. When the first byte of output $\mathsf{X}[0]$ is produced, $j_{257} = 3$ and $\mathsf{S}_{256}[4] = 117$ is swapped with $\mathsf{S}_{257}[1]$. $\mathsf{S}_{257}[1] = 117$ remains unmodified for the next 255 steps of the RC4-PRGA. When $\mathsf{X}[256]$ is produced, $j_{513} = 221$ and $\mathsf{S}_{512}[1] = 117$ and $\mathsf{S}_{512}[140] = 166$ are swapped. The output is now $\mathsf{S}_{513}[\mathsf{S}_{513}[1] + \mathsf{S}_{513}[176]] = \mathsf{S}_{513}[1] = 140$. Figures 6.16 and 6.17 illustrate these steps.

An attacker who calculates

$$\begin{aligned}\mathcal{F}_{Mantin}(28,230,191,140) &= \mathsf{S}_3^{-1}[1-\mathsf{X}[256]] - j_3 - \mathsf{S}_3[3] \\ &= \mathsf{S}_3^{-1}[1-140] - 140 - 1 \\ &= \mathsf{S}_3^{-1}[117] - 140 - 1 \\ &= 117 - 140 - 1 \\ &= 232 \\ &= \mathsf{K}[3]\end{aligned} \tag{6.28}$$

would have gotten the correct key byte $\mathsf{K}[3]$.

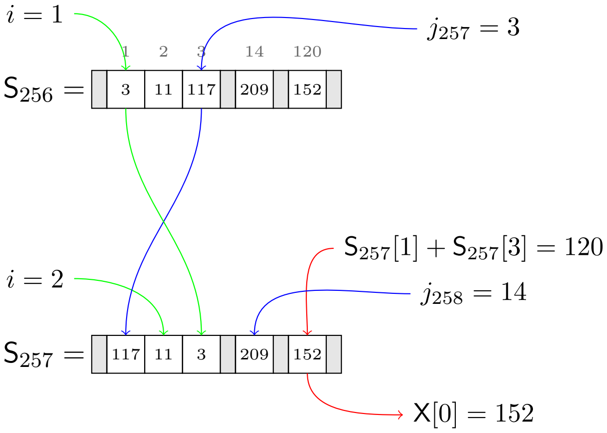


Figure 6.16: First byte of output for $\mathsf{K} = 221, 37, 135, 232, 150, 200, 133, 253$

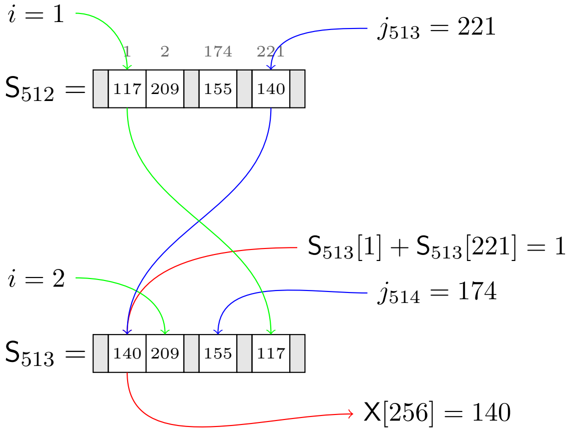


Figure 6.17: 256th byte of output for $\mathsf{K} = 221, 37, 135, 232, 150, 200, 133, 253$

6.3.4 Implementation

An implementation of *Mantin's second round attack* is not available in the *aircrack-ng toolsuite*. The attack performs much worse than any other attack and the first two bytes of the key stream are nearly always guessable in a WEP network. This attack is only described because it was the first key recovery attack against WEP which uses the *Jenkins' correlation*.

7 The PTW attack

Even if Mantin's attack from ASIACRYPT'05 [Man05] was the first key recovery attack against WEP, which did use the Jenkins' correlation (Section 6.3), it is not the most effective attack you can build using the Jenkins' correlation. Again, all additions and subtractions in this Chapter, except for probabilities, are done mod n.

7.1 Klein's Analysis on RC4

In 2005, Andreas Klein gave a first talk about his analysis of RC4. In a nutshell, Klein found two attacks on RC4 in WEP-like modes of operations.

- The first attack is the same as Mantin's attack from ASIACRYPT'05 [Man05]. Additionally, Klein analyzed the case, where the initialization vector is appended to the *root key* ($\mathsf{Rk}||\mathsf{IV}$). In WEP, the initialization vector is always prepended to the *root key* ($\mathsf{IV}||\mathsf{Rk}$). Klein used a slightly different theoretical model to analyze the success probability for the attack, which is very close to the model I used in Section 6.3.2.
- The second attack is a first round attack as the *FMS attack* or *KoreK attack*. Instead of using just the first two bytes of output of the RC4-PRGA, $\mathsf{X}[k]$ is used to determine the key byte $\mathsf{K}[k+1]$.

I have already discussed the second round attack in Section 6.3.2. For the rest of this chapter, I will only focus on Klein's first round attack.

7.1.1 Klein's first round attack

Klein's first round attack is the simplest and most pretty attack on RC4 in WEP-likes modes of operations, I am currently aware of. Basically, Klein could show the following:

Attack 8 *Klein (2005): An attacker who has access to an oracle $O_{WEP}(3, 13, 15)$ can recover the secret key of O_{WEP} with a success probability of 50% with 43,000 queries to O_{WEP} and negligible computational effort. The same holds if the attacker has access to O_{WEP_CTR} or O_{WEP_LINUX}. The attacker can recover the secret key with a success probability of 95% with 70,000 queries to O_{WEP}, O_{WEP_CTR} or O_{WEP_LINUX}.*

Mathematical background

The idea behind Klein's attack is quite simple but effective. Let's assume that we know $\mathsf{K}[0] \dots \mathsf{K}[l-1]$ and want to recover $\mathsf{K}[l]$. We can simulate the first l steps of the RC4-KSA. In the next step, the value $k = \mathsf{S}_l[j_l + \mathsf{S}_l[l] + \mathsf{K}[l]]$ is swapped to $\mathsf{S}_{l+1}[l]$. Knowledge of k would reveal $\mathsf{K}[l]$, as it does in the *FMS attack* or *Mantin's second round attack*. i will only take the value l again after exactly $n-2$ steps of the remaining RC4-KSA ($n-l-1$ steps are executed here) and the RC4-PRGA ($l-1$ steps are executed, before i takes the value l again). So $\mathsf{S}_{l+1}[l]$ will only be changed if it is hit by j, which happens with a probability of exactly $\left(\frac{1}{n}\right)^{n-2}$ in the generalized randomized RC4-KSA and RC4-PRGA. With a probability of $1 - \left(\frac{1}{n}\right)^{n-2}$, $\mathsf{S}_{l+1}[l]$ will be modified and set to a different value than k.

We now make use of the Jenkins' correlation. We have to distinguish between two cases:

1. $\mathsf{S}_{n+l-1}[l] = k$

   This happens with probability $\left(\frac{1}{n}\right)^{n-2}$. According to Jenkins' correlation, the probability for $l - \mathsf{X}[l-1] = k$ in this case is:

   $$\text{Prob}\left(l - \mathsf{X}[l-1] = k \mid \mathsf{S}_{n+l-1}[l] = k\right) = \frac{2}{n} \tag{7.1}$$

2. $\mathsf{S}_{n+l-1}[l] \neq k$

   This happens with probability $1 - \left(\frac{1}{n}\right)^{n-2}$. According to Jenkins' correlation, the probability for $\mathsf{X}[l-1] = k$ in this case is:

   $$\text{Prob}\left(l - \mathsf{X}[l-1] = k \mid \mathsf{S}_{n+l-1}[l] \neq k\right) = \frac{n-2}{n(n-1)} \tag{7.2}$$

In total, this gives us the following probability for $\mathsf{X}[l-1] = k = \mathsf{S}_l[j_l + \mathsf{S}_l[l] + \mathsf{K}[l]]$:

$$\text{Prob}\left(l - \mathsf{X}[l-1] = k\right) = \left(\left(\frac{1}{n}\right)^{n-2}\right)\frac{2}{n} + \left(1 - \left(\frac{1}{n}\right)^{n-2}\right)\frac{n-2}{n(n-1)} \tag{7.3}$$

Solving this for $\mathsf{K}[l]$ gives us the following formula:

$$\mathcal{F}_{Klein}(\mathsf{K}[0], \dots, \mathsf{K}[l-1], \mathsf{X}[l-1]) = \mathsf{S}_l^{-1}[l - \mathsf{X}[l-1]] - (\mathsf{S}_l[l] + j_l) \tag{7.4}$$

For $n = 256$, the probability for $\mathcal{F}_{Klein}$ taking the value of $\mathsf{K}[l]$ is approximately $\frac{1.3676}{n}$. This probability does not depend on l, because there are always $n-2$ swaps in the RC4-KSA and RC4-PRGA, before i takes the value l again.

If we look at the *Klein attack* compared to the *FMS* or *KoreK attack*, we should notice some interesting differences.

- The *Klein attack* is able to make use out of every single session we receive from O_{WEP}. Even if the probability that $\mathcal{F}_{FMS}$ does return a correct value is much higher than the probability that $\mathcal{F}_{Klein}$ does return a correct value, the *Klein attack* totally outperforms the *FMS attack*, because it is able to collect information from every single session. The *KoreK attack* is able to use more sessions than *FMS attack*, but is still much worse than the *Klein attack*, when it comes to recover a key with a relatively low number of sessions.

- The success probability of $\mathcal{F}_{Klein}$ does not depend on the key byte which is attacked. Both *FMS* and *KoreK* had lower success probabilities for the first key bytes than for the last key bytes. Additionally, the total percentage of usable sessions is much lower for the first key bytes than for the last key bytes.

- The *Klein attack* uses more bytes of the key stream. The exact number ob bytes needed by *Klein* depends on the length of the secret *root key* and the length of the *IV value*. This makes it harder to apply, if a method for key stream recovery is used, which can not recover all needed key stream bytes with a high enough certainty. We will discuss that problem later.

Unfortunately, Klein did never implement or tested his attack against a real WEP secured network.

7.1.2 An example

Let's assume that RC4 is used with $\mathsf{K} = 12, 111, 28, 107, 226, 211, 232, 247$. The attacker knows the first $l = 3$ bytes of K and is interested in $\mathsf{K}[l]$.

The attacker can simulate the first 3 steps of the RC4-KSA. In the next step

$$\begin{aligned} j_4 &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] \\ &= 154 + 3 + 107 \\ &= 8 \end{aligned} \tag{7.5}$$

$\mathsf{S}_3[3] = 3$ and $\mathsf{S}_3[8] = 8$ are now swapped and $\mathsf{S}_4[3] = 8$. $\mathsf{S}[4]$ remains unchanged for the rest of the RC4-KSA and the beginning of the RC4-PRGA. These steps of the RC4-KSA are illustrated in figure 7.1.

When $\mathsf{X}[2]$ is generated, $j_{n+3} = 182$ and $\mathsf{X}[2] = \mathsf{S}_{n+3}[\mathsf{S}_{n+3}[3] + \mathsf{S}_{n+3}[182]] = 251$ is the output of the RC4-PRGA. An attacker who calculates

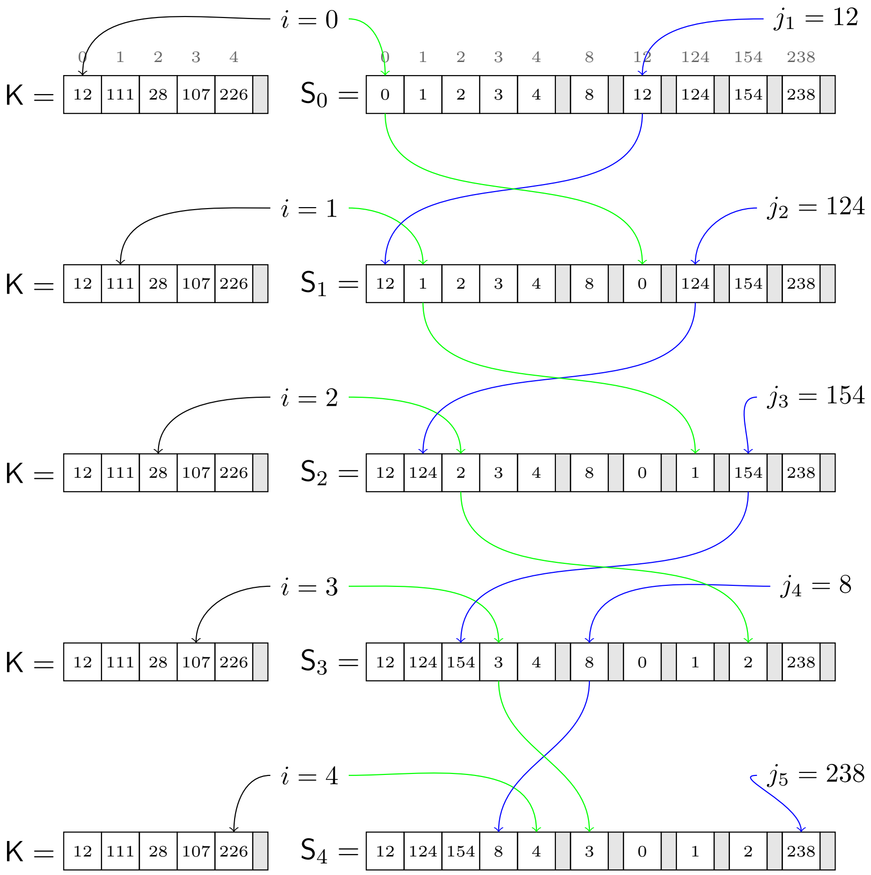


Figure 7.1: First 4 steps of RC4-KSA for $\mathsf{K} = 12, 111, 28, 107, 226, 211, 232, 247$

$$\begin{aligned}
\mathcal{F}_{Klein}(12, 11, 28, 251) &= \mathsf{S}_3^{-1}[3 - \mathsf{X}[2]] - (\mathsf{S}_3[3] + j_3) \\
&= \mathsf{S}_3^{-1}[3 - 251] - (3 + 154) \\
&= \mathsf{S}_3^{-1}[8] - 157 \\
&= 8 - 157 \\
&= 107 \\
&= \mathsf{K}[3]
\end{aligned} \tag{7.6}$$

would have gotten the right value for $\mathsf{K}[3] = 107$.

7.1.3 Implementation

Aircrack-ng contains an implementation of the *Klein attack*, which does not do key ranking. All implementations of previous attacks do key ranking, but where limited to 3 minutes of CPU time at most in all benchmarks to esti-

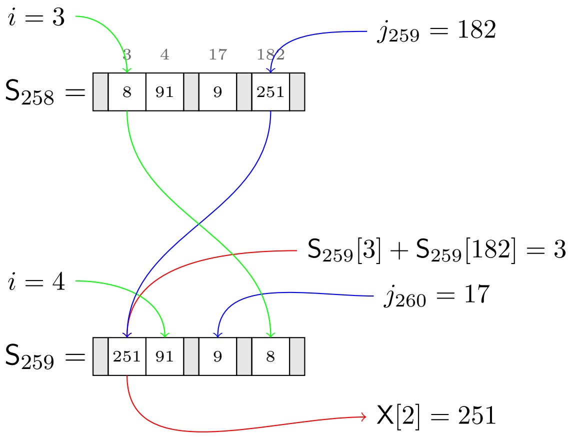
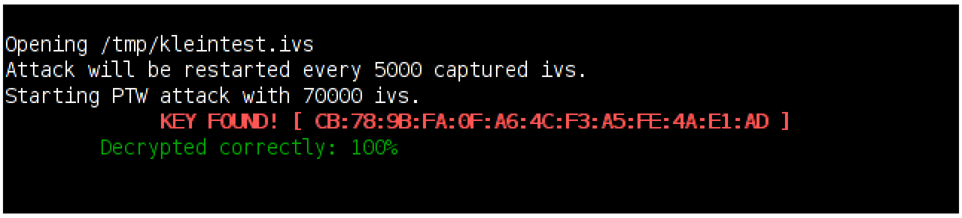


Figure 7.2: Generation of $X[2]$ for $K = 12, 111, 28, 107, 226, 211, 232, 247$



```
Opening /tmp/kleintest.ivs
Attack will be restarted every 5000 captured ivs.
Starting PTW attack with 70000 ivs.
             KEY FOUND! [ CB:78:9B:FA:0F:A6:4C:F3:A5:FE:4A:E1:AD ]
        Decrypted correctly: 100%
```

Figure 7.3: aircrack-ng 1.0 beta 1 Klein results

mate their success rate. To launch a *Klein attack* on all packets captured in `/tmp/kleintest.ivs`, an attacker has to execute the following command:

```
./aircrack-ng -0 -p 1 -P 2 /tmp/kleintest.ivs
```

If the attack was successful, an output similar to figure 7.3 will be displayed.

Because no key ranking is done, no progress is displayed by aircrack while calculating the key.

7.1.4 Success rate

To measure the success rate for the *Klein attack*, aircrack-ng was used. It usually takes less than a second to execute the *Klein attack*. The results are shown in figure 7.4.

As you can see, the *Klein attack* reaches nearly 100% success probability.

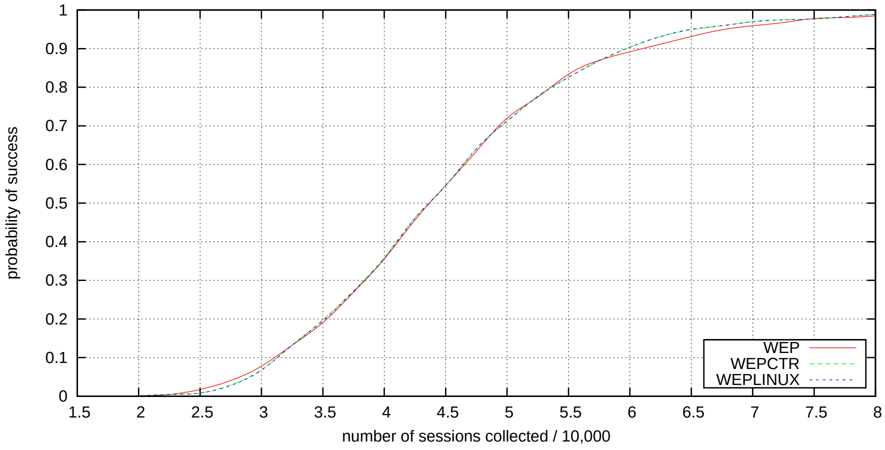


Figure 7.4: Klein success rate

7.2 Key ranking

Until now, all attacks we have seen so far, somehow tried to guess an unknown value $\mathsf{K}[l]$, of a RC4 key, using a lot of different sessions with the same value for $\mathsf{K}[l]$, and with $\mathsf{K}[0]$ to $\mathsf{K}[l-1]$ known to the attacker. The method used for deciding on $\mathsf{K}[l]$ is always a voting process, where all sessions available to the attacker or a subset of them are examined and every session can vote for $\mathsf{K}[l]$ having certain values, or in the case of the *A_neg KoreK attack*, not having a certain value. After all votes have been accounted, a guess or let's say a *decision* for $\mathsf{K}[l]$ was made. Now $\mathsf{K}[l]$ can be treated as known and the attacker can continue to recover $\mathsf{K}[l+1]$. From an abstract point of view, this can be seen as a kind decision tree for the secret *root key*.

The first decision needs to be made for $\mathsf{Rk}[0]$. Usually we assume that the value with the most votes will be the correct value for $\mathsf{Rk}[0]$ and continue with the determination of $\mathsf{Rk}[1]$. This works great if a high number of sessions are available to the attacker. Unfortunately if the number of sessions is relatively low, the correct value for $\mathsf{Rk}[0]$ is not necessarily the most voted value, but tends to be one of the top voted values. Let's assume that the most voted value is not $\mathsf{Rk}[0]$, but $\mathsf{Rk}[0]$ is the second most voted value.

A simple implementation of an attack, which always assumes that the most voted value is the correct value, would fail, because $\mathsf{Rk}[0]$ is incorrect and therefore the computed key will be incorrect. Of course, an attacker would still like to be able to compute the key with a lower amount of sessions available. Instead of computing a single key, an attacker can compute a set of keys, by following some alternative computations paths too. For example, an attacker could just assume that $\mathsf{Rk}[0]$ was the first or second most voted value and follow both computational paths. By then testing every single key in this set for correctness, an attacker can find the correct key in this set. Usually, such an approach is called *key ranking*, which comes from *linear cryptanalysis* [Mat94] or *error correction*, which comes from the area of *statistical attacks* [Sch00].

7.2.1 Key ranking strategies

Of course, an attacker can usually not follow all possible alternative paths, which would be equivalent to a brute force attack on the whole key space. There are multiple strategies to decide which alternative computational paths to follow. I will present some ideas here:

Static number of paths to follow: This is a very simple strategy. At each decision point, the k most probable ways are followed. This is very easy to implement, but has some disadvantages. First, the number of key candidates cannot really be fine adjusted. In general, if there are l_{key} decisions to be made, and k is the number of paths to follow at every single decision, this strategy will compute

$$k^{l_{key}} \tag{7.7}$$

different keys and

$$\frac{k^{l_{key}+1}-1}{k-1} \tag{7.8}$$

different voting processes will happen during these computations. If the secret *root key* has a length of $l_{key} = 13$ bytes, the only useful values for k are 2, which results in $2^{13} = 8,192$ different keys and $16,383$ voting processes, or 3, which results in $3^{13} = 1,594,323$ different keys and $2,391,484$ voting processes. Even for $k = 3$, the number of required voting processes is so high, that this value might be unfeasible on an average desktop computer.

Static number of alternative choices per path: Let's assume we just take the most voted value and the second most voted value as possible choices. We can additionally demand, that every computational path is allowed to take the second value ad most k times. In total, this will result in

$$\sum_{m=0}^{k} \binom{l_{iv}}{m} \tag{7.9}$$

different keys and

$$\sum_{m=1}^{k+1} \binom{l_{iv}}{m} \tag{7.10}$$

voting processes during the whole computation. For $k = l_{iv}$, this method is equivalent to the previous one, with 2 paths to follow.

This scheme can be extended to taking not just the second candidate, but other top voted values.

Intelligent choosing of alternate computation paths: Both methods presented so far just use the order of the possible values for each key byte. It is possible to use more information from the voting process. Let's assume that an attacker has tried a simple attack, but at the end of the computation path, the resulting key is incorrect. This means that at least one of the decisions in the path was incorrect. An attacker can now start looking for a decision where the difference between the chosen value and the next alternative candidate was relatively low. Naturally speaking, the attacker is looking for a decision, where he was relatively uncertain about which value to decide for. At this point, the next best alternate candidate is chosen and the attack is continued at this point.

This is only the basic idea behind this strategy. Unfortunately, if a decision $\mathsf{Rk}[l] = c$ was wrong, all following voting processes will be based on the assumption that $\mathsf{Rk}[l] = c$, and will perhaps just result in more or less random values. An attacker would now mostly choose a decision after $\mathsf{Rk}[l]$, because the difference between the number of votes tends to be small, if the voting process just outputs random votes. So instead of choosing the decision the attacker was most uncertain about, an attacker should choose the first decision in the computation path he was somehow uncertain about.

Aircrack-ng implements such a strategy for the *FMS* and *KoreK attack*. At every decision point in the decision tree, aircrack will try all values which got at least half of the votes of the most voted value. This behavior can be controlled by the so called *fudge factor*. Setting the *fudge factor* to k makes aircrack try all values which got at least $\frac{1}{k}$ of the votes of the most voted value.

7.2.2 General improvements for key ranking

Additionally, there are some strategies which can be used nearly independently of the key ranking strategy, to improve the effectiveness of the attack:

Key space restriction: We assume that the attacker has some knowledge of the key which is being used. For example, some vendors ship access points with default keys which just consist of ASCII characters representing decimal numbers. This means that every $\mathsf{Rk}[l]$ will have a value between 48 and 57 inclusively. At every decision point where a decision would be made, that would result not in $48 \leq \mathsf{Rk}[l] \leq 57$, we choose the next best alternative value, which results in a valid value for $\mathsf{Rk}[l]$.

Aircrack implements such a mode, by specifying the parameter `-h` to aircrack, it will only allow values in the key, which represent numbers in the ASCII code. By specifying the parameter `-c`, aircrack will only allow alpha-numeric key bytes. By specifying the parameter `-t`, aircrack will only allow binary coded decimal key bytes.

Brute forcing the last bytes: Because a voting process is much more expensive in CPU time than testing a small amount of keys, an attacker can skip all voting processes for the last one or two bytes, and just try all possible values for them.

Aircrack does a brute force search for the last key byte by default for the *FMS* and *KoreK attack*. By specifying the parameter `-x2`, a brute force search is done on the last two bytes of the key.

Brute forcing the first byte: Most attacks are much better in determine the correct value for the last key bytes than the first key byte. An attacker might choose to just try all possible values for the first key byte, instead of determine it using a voting process. The drawback of such an approach would be, that it will heavily increase the CPU time by at most factor 256.

7.3 The basic PTW attack

Even if the *Klein attack* is a very simple and effective attack when compared to the *FMS attack* or the *KoreK attack*, there is still some room for improvements. Unfortunately, we will have to give up some of the simplicity.

Attack 9 *Pyshkin, Tews, Weinmann (2007): An attacker who has access to an oracle $O_{WEP}(3, 13, 15)$ can recover the secret key of O_{WEP} with a success probability of 50% with 35,000 queries to O_{WEP} and negligible computational effort. The same holds if the attacker has access to O_{WEP_CTR} or O_{WEP_LINUX}. The attacker can recover the secret key with a success probability of 95% with 55,000 queries to O_{WEP}, O_{WEP_CTR} or O_{WEP_LINUX}.*

7.3.1 Determine sums of key bytes instead of key bytes

The basic idea is simple. If an attacker can determine $\mathsf{Rk}[0]$ and $\mathsf{Rk}[0] + \mathsf{Rk}[1] \bmod n$ independently, he can calculate $\mathsf{Rk}[0]$ and $\mathsf{Rk}[1] = (\mathsf{Rk}[0] + \mathsf{Rk}[1]) - \mathsf{Rk}[0]$ from these values. If the attacker later decides that the decision for $\mathsf{Rk}[0]$ was incorrect, he just has to do a new single subtraction to update the value for $\mathsf{Rk}[1]$. For the rest of this document, let $\sigma_i = \sum_{m=0}^{i} \mathsf{Rk}[m]$.

In general, a attacker who knows all values σ_0 to $\sigma_{l_{key}-1}$ can compute $\mathsf{Rk}[0]$ to $\mathsf{Rk}[l_{key} - 1]$ from these values. Only l_{key} subtractions mod n are needed for these computations, which is relatively low to the computational effort of an single RC4-KSA, which usually needs at least $2 \cdot n$ additions mod n, depending on the exact implementation. Of course, it is possible to compute $\mathsf{Rk}[0]$

to $\mathsf{Rk}[l_{key}-1]$ from σ_0 to $\sigma_{l_{key}-1}$, using just l_{key} additions mod n. So an attacker can alternatively try to determine σ_0 to $\sigma_{l_{key}-1}$, and test these values for correctness, without any noticeable impact in the performance.

7.3.2 Extending the Klein attack to the sum of the first 2 key bytes

Let's recall the Klein attack. We assume again that the attacker knows $\mathsf{K}[0]$ to $\mathsf{K}[l-1]$. Now, the attacker is interested in $\mathsf{K}[l]+\mathsf{K}[l+1]$ mod n instead of $\mathsf{K}[l]$. In step $l+1$ of the RC4-KSA, $S_l[l]+\mathsf{K}[l]$ is added to j_l and $\mathsf{S}_l[j_{l+1}] = k$ and $\mathsf{S}_l[l]$ are swapped. In the Klein attack, we used Jenkins' correlation to guess k from $\mathsf{X}[l-1]$ and then determined $\mathsf{K}[l]$ from k.

Now, we ignore this step and look at step $l+2$ of the RC4-KSA. Here $\mathsf{S}_{l+1}[l+1]+\mathsf{K}[l+1]$ is added to j_{l+1} and we got

$$\begin{aligned} j_{l+2} &= j_{l+1} + \mathsf{S}_{l+1}[l+1] + \mathsf{K}[l+1] \\ &= j_l + \mathsf{S}_l[l] + \mathsf{K}[l] + \mathsf{S}_{l+1}[l+1] + \mathsf{K}[l+1] \end{aligned} \tag{7.11}$$

Now $\mathsf{S}_{l+1}[l+1]$ and $\mathsf{S}_{l+1}[j_{l+2}] = k'$ are swapped. Let's assume that $\mathsf{S}_{l+1}[l+1] = \mathsf{S}_l[l+1]$ and $\mathsf{S}_{l+1}[j_{l+2}] = \mathsf{S}_l[j_{l+2}]$ holds. (We will estimate the exact probability for that later) If the attacker knows

$$\mathsf{S}_{l+2}[l+2] = \mathsf{S}_l[j_l + \mathsf{S}_l[l] + \mathsf{K}[l] + \mathsf{S}_l[l+1] + \mathsf{K}[l+1]] \tag{7.12}$$

he can compute $\mathsf{K}[l]+\mathsf{K}[l+1]$ mod n by solving equation 7.12 for $\mathsf{K}[l]+\mathsf{K}[l+1]$. Again, we can use the Jenkins' correlation to determine $\mathsf{S}_{l+2}[l+2]$ from $\mathsf{X}[l+1]$. In total, that gives us the following function

$$\mathcal{F}_{ptw_2}(\mathsf{K}[0],\ldots,\mathsf{K}[l-1],\mathsf{X}[l]) = \mathsf{S}_l^{-1}[l+1-\mathsf{X}[l]] - (j_l + \mathsf{S}_l[l] + \mathsf{S}_l[l+1]) \tag{7.13}$$

We will now try to estimate the success probability that $\mathcal{F}_{ptw_2}$ takes the correct value $\mathsf{K}[l]+\mathsf{K}[l+1]$ mod n. As in the Klein attack, we have two cases:

- The following conditions are met:
  1. $\mathsf{S}_{l+1}[l+1] = \mathsf{S}_l[l+1]$
     This can be expressed as j_{l+1} does not take the value $l+1$. This happens with probability $\frac{n-1}{n}$.
  2. $\mathsf{S}_{l+1}[j_{l+2}] = \mathsf{S}_l[j_{l+2}]$
     This can be expressed as j_{l+1} and i do not take the value j_{l+2}. We

assume that this happens independently of the previous condition and happens with a probability of $\left(\frac{n-1}{n}\right)^2$.

3. $\mathsf{S}_{l+2}[l+2]$ stays unmodified in the next $n-2$ steps.

   As in the Klein attack, this happens with probability $\left(\frac{n-1}{n}\right)^{n-2}$.

In total, the probability that $\mathcal{F}_{ptw_2}$ takes the value $\mathsf{K}[l] + \mathsf{K}[l+1] \bmod n$ and these 3 conditions are met is:

$$\left(\frac{n-1}{n}\right)^3 \cdot \left(\frac{n-1}{n}\right)^{n-2} \cdot \frac{2}{n} \tag{7.14}$$

The part $\left(\frac{n-1}{n}\right)^3$ can be seen as a correction for the original success probability of $\mathcal{F}_{Klein}$.

- One of these three conditions is not met.

  If just condition 3 is not met, $\mathsf{S}_{l+n}[l+1]$ will definitely not contain k'. If condition 1 or 2 is not met, $\mathsf{S}_{l+n}[l+1]$ might contain k'. But we can say, that the probability that $\mathcal{F}_{ptw_2}$ takes the value of $\mathsf{K}[l] + \mathsf{K}[l+1] \bmod n$ if at least one of the three conditions is not met is at least:

$$\left(1 - \left(\frac{n-1}{n}\right)^3 \cdot \left(\frac{n-1}{n}\right)^{n-2}\right) \cdot \frac{n-2}{n(n-1)} \tag{7.15}$$

In total, the probability that $\mathcal{F}_{ptw_2}$ takes the correct value is at least:

$$q_2 \cdot \left(\frac{n-1}{n}\right)^{n-2} \cdot \frac{2}{n} + \left(1 - q_2 \cdot \left(\frac{n-1}{n}\right)^{n-2}\right) \cdot \frac{n-2}{n(n-1)} \tag{7.16}$$

with

$$q_2 = \left(\frac{n-1}{n}\right)^3 \tag{7.17}$$

q_2 can be seen as a kind of correction factor for the success probability of the *Klein attack* which takes into account, that we need certain things not to happen in the two steps of the RC4-KSA, until k' is swapped to its location.

An example

Let's assume that RC4 is used with $\mathsf{K} = 110, 106, 205, 97, 83, 37, 81, 179$. The attacker knows the first $l = 3$ bytes of K and is interested in $\mathsf{K}[3] + \mathsf{K}[4]$.

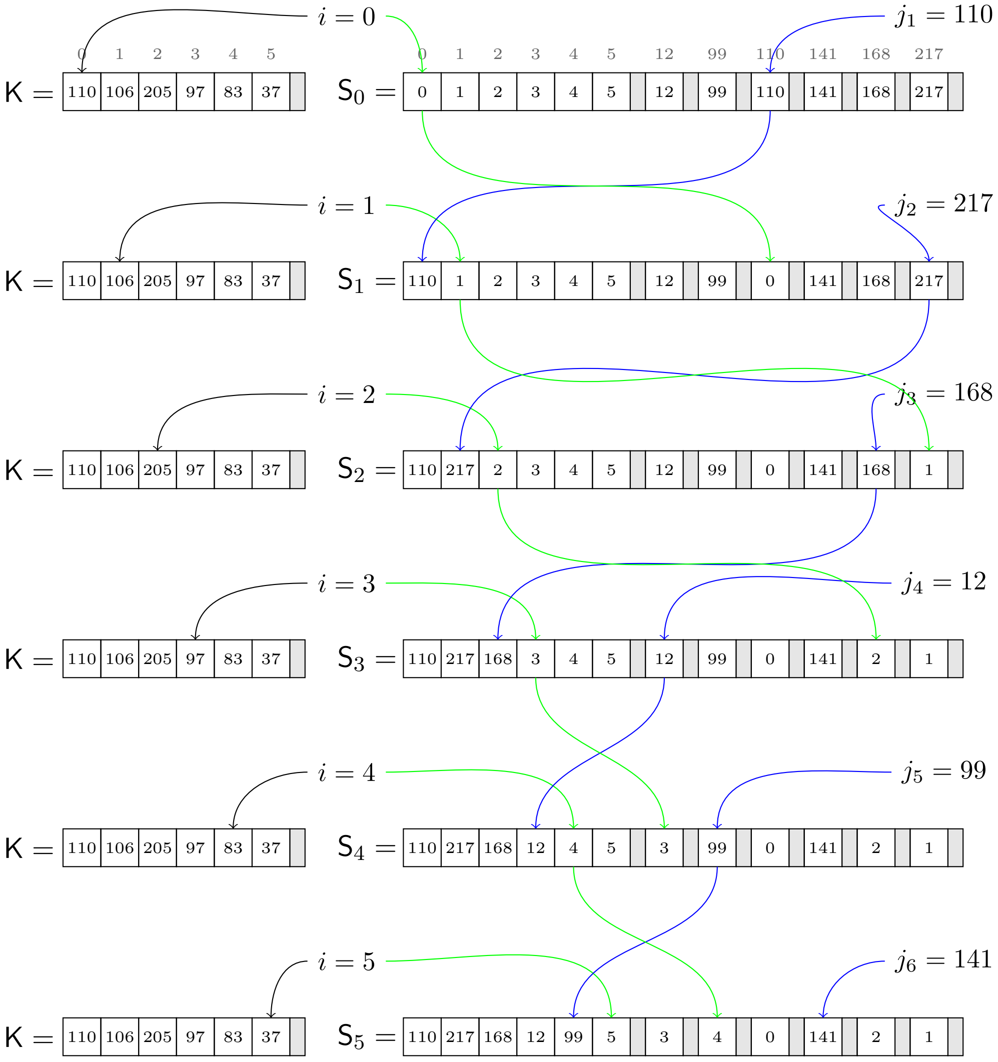

$i = 0$
$j_1 = 110$
0 1 2 3 4 5
0 1 2 3 4 5 12 99 110 141 168 217
$\mathsf{K} =$ 110 106 205 97 83 37
$\mathsf{S}_0 =$ 0 1 2 3 4 5 12 99 110 141 168 217
$i = 1$
$j_2 = 217$
$\mathsf{K} =$ 110 106 205 97 83 37
$\mathsf{S}_1 =$ 110 1 2 3 4 5 12 99 0 141 168 217
$i = 2$
$j_3 = 168$
$\mathsf{K} =$ 110 106 205 97 83 37
$\mathsf{S}_2 =$ 110 217 2 3 4 5 12 99 0 141 168 1
$i = 3$
$j_4 = 12$
$\mathsf{K} =$ 110 106 205 97 83 37
$\mathsf{S}_3 =$ 110 217 168 3 4 5 12 99 0 141 2 1
$i = 4$
$j_5 = 99$
$\mathsf{K} =$ 110 106 205 97 83 37
$\mathsf{S}_4 =$ 110 217 168 12 4 5 3 99 0 141 2 1
$i = 5$
$j_6 = 141$
$\mathsf{K} =$ 110 106 205 97 83 37
$\mathsf{S}_5 =$ 110 217 168 12 99 5 3 4 0 141 2 1

Figure 7.5: First 5 steps of RC4-KSA for $\mathsf{K} = 110, 106, 205, 97, 83, 37, 81, 179$

After the first 3 steps of the RC4-KSA,

$$\begin{aligned} j_4 &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] \\ &= 168 + 3 + 97 \\ &= 12 \end{aligned} \tag{7.18}$$

Now $\mathsf{S}_3[3] = 3$ and $\mathsf{S}_3[12] = 12$ are swapped. In the next step:

$$\begin{aligned} j_5 &= j_4 + \mathsf{S}_4[4] + \mathsf{K}[4] \\ &= 12 + 4 + 83 \\ &= 99 \\ &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] + \mathsf{S}_4[4] + \mathsf{K}[4] \\ &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] + \mathsf{S}_3[4] + \mathsf{K}[4] \end{aligned} \tag{7.19}$$

and $\mathsf{S}_4[4] = 4$ and $\mathsf{S}_4[99] = 99$ are swapped. $\mathsf{S}_5[4] = 99$ remains unchanged for the rest of the RC4-KSA and the beginning of the RC4-PRGA. These steps are illustrated in figure 7.5.

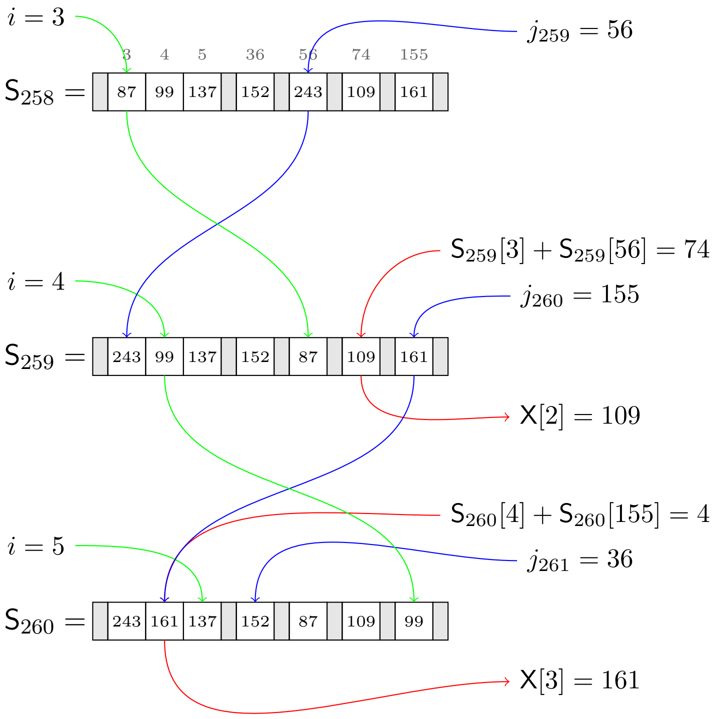


Figure 7.6: Generation of $\mathsf{X}[2]$ and $\mathsf{X}[3]$ for $\mathsf{K} = 110, 106, 205, 97, 83, 37, 81, 179$

When $\mathsf{X}[3]$ is produced, $j_{n+4} = 155$ and $\mathsf{S}_{n+3}[4] = 99$ and $\mathsf{S}_{n+3}[155] = 161$ are swapped. The output of the RC4-PRGA is now

$$\begin{aligned} \mathsf{S}_{n+4}[\mathsf{S}_{n+4}[4] + \mathsf{S}_{n+4}[155]] &= \mathsf{S}_{n+4}[161 + 99] \\ &= \mathsf{S}_{n+4}[4] \\ &= 161 \end{aligned} \tag{7.20}$$

An attacker who calculates

$$\begin{aligned}
&\mathcal{F}_{ptw_2}(110, 106, 205, 161)\\
=&\mathsf{S}_3^{-1}[3 + 1 - \mathsf{X}[2]] - (j_3 + \mathsf{S}_3[3] + \mathsf{S}_3[3 + 1])\\
=&\mathsf{S}_3^{-1}[4 - 161] - (168 + 3 + 4)\\
=&\mathsf{S}_3^{-1}[99] - 175\\
=&99 - 175\\
=&180\\
=&97 + 83\\
=&\mathsf{K}[3] + \mathsf{K}[4]
\end{aligned} \tag{7.21}$$

would have gotten the correct sum $\mathsf{K}[3]+\mathsf{K}[4] = 180$. These steps are illustrated in figure 7.6.

7.3.3 Extending the Klein attack to sums of key bytes

Using the same method, we can find functions, which take the value of $\sum_{a=l}^{m} \mathsf{K}[a] \bmod n$ with a slightly lower probability than $\mathcal{F}_{Klein}$ takes the value of the next key byte $\mathsf{K}[l]$.

In general, in the next m steps of the RC4-KSA, the value

$$\sum_{a=l}^{l+m-1} (\mathsf{S}_a[a] + \mathsf{K}[a]) \bmod n \tag{7.22}$$

is added to j_l. For example, for $m = 3$, this is

$$\mathsf{S}_l[l] + \mathsf{S}_{l+1}[l + 1] + \mathsf{S}_{l+2}[l + 1] + \mathsf{K}[l] + \mathsf{K}[l + 1] + \mathsf{K}[l + 2] \tag{7.23}$$

Now, $\mathsf{S}_{l+m-1}[j_{l+m}] = k$ is swapped with $\mathsf{S}_{l+m-1}[l + m - 1]$. The result is stored in $\mathsf{S}_{l+m}[l + m - 1] = k$. If the attacker can determine $\mathsf{S}_{l+m}[l + m - 1]$ and knows the location of k in S_{l+m-1}, he can determine what was added to j_l in the next m steps of the RC4-KSA. If none of the value from $\mathsf{S}_{l+1}[l+1]$ to $\mathsf{S}_{l+m-1}[l+m-1]$ has been altered, before it was added to j, this will reveal $\sum_{a=l}^{l+m-1} \mathsf{K}[a] \bmod n$. With other words, if we assume that $\mathsf{S}_l[j_{l+m}] = \mathsf{S}_{l+m-1}[j_{l+m}] = k$ and $j_{l+m} = j_l + \sum_{a=l}^{l+m-1} (\mathsf{S}_l[a] + \mathsf{K}[a])$ holds, we can solve this for $\sum_{a=l}^{l+m-1} \mathsf{K}[a]$ and get the following formula:

$$\sum_{a=l}^{l+m-1} \mathsf{K}[a] = \mathsf{S}_l^{-1}[k] - j_l - \left(\sum_{a=l}^{l+m-1} \mathsf{S}_l[a] \right) \tag{7.24}$$

Using the *Jenkins' correlation*, we know that $l + m - 1\mathsf{X}[l + m - 2]$ tends to take the value of k. In total, this gives us the following functions:

$$\begin{aligned}&\mathcal{F}_{ptw_m}(\mathsf{K}[0],\ldots,\mathsf{K}[l-1],\mathsf{X}[l+m-2])\\ =&\mathsf{S}_l^{-1}[l+m-1-\mathsf{X}[l+m-2]]-\left(\sum_{a=l}^{l+m-1}\mathsf{S}_l[a]\right)\end{aligned}\tag{7.25}$$

An estimation of the success probability

We will now try to get a good lower bound for the probability of $\mathcal{F}_{ptw_m}$ to take the correct value. We will extend the special case for two bytes we have seen before to an arbitrary number m of following key bytes.

We will use a similar approach to the special case with just the sum of two key bytes. We will try to estimate, with which probability certain events in the RC4-KSA will occur:

- $\sum_{a=l}^{l+m-1}\mathsf{S}_a[a]=\sum_{a=l}^{l+m-1}\mathsf{S}_l[a]$

  These sums will at least be equal, if all summands are equal. This means that j is not allowed to modify $\mathsf{S}[a]$ after j_l and before the value $\mathsf{S}_a[a]$ is added to j in the $a+1$th step. In the $l+1$th step, j_{l+1} is not allowed to take any value from $l+1$ to $l+m-1$ inclusively, which happens with probability $\left(\frac{n-m+1}{n}\right)$.

  In the next step, j_{l+2} is allowed to take the value $l+1$, because $\mathsf{S}_{l+1}[l+1]$ has already been added to j_{l+1}, but no value from $l+2$ to $l+m-1$ inclusively. In general, j_{l+h} is not allowed to take exactly $m-h$ different values. For a single step, this happens with probability $\frac{n-(m-h)}{n}$. For all steps from step $l+1$ to step $l+m$, this happens with probability

$$\prod_{a=1}^{m-1}\left(\frac{n-a}{n}\right)\tag{7.26}$$

  There is still a small chance that multiple summands are unequal, but $\sum_{a=l}^{l+m-1}\mathsf{S}_a[a]=\sum_{a=l}^{l+m-1}\mathsf{S}_l[a]$ holds anyway. Because we are just interested in a good lower bound for the success probability, we ignore this, to keep our formulas a little bit more simple.

- $\mathsf{S}_l[k]=\mathsf{S}_{l+m-1}[k]$

  Now, $\mathsf{S}_l[k]$ could be modified by i or j in any of the following $m-1$ steps, before $\mathsf{S}[k]$ is swapped to $\mathsf{S}_{l+m}[l+m-1]$. If j changes randomly, j will not take the value k with probability $\left(\frac{n-1}{n}\right)^{m-1}$. Because i acts as a counter, it will take exactly $m-1$ different values in the next $m-1$ steps. The probability that none of these values is k is $\left(\frac{n-(m-1)}{n}\right)$. In total, the probability that $\mathsf{S}_l[k]=\mathsf{S}_{l+m-1}[k]$ holds is at least:

$$\left(\frac{n-1}{n}\right)^{m-1}\left(\frac{n-(m-1)}{n}\right) \tag{7.27}$$

This is again a lower bound, because if i and j hit $\mathsf{S}[k]$ in the same step, $\mathsf{S}[k]$ will not be modified at all.

If both conditions are holding, the value $\mathsf{S}_{l+m}[l+m-1]$ will not be modified in the remaining $n-2$ steps of the RC4-KSA and RC4-PRGA, until $\mathsf{X}[l+m-2]$ is produced. With a probability of $\frac{2}{n}$, $\mathcal{F}_{ptw_m}$ will take the correct value of $\sum_{a=l}^{l+m-1}\mathsf{K}[a]$. If one of these conditions is not holding, most probably a wrong value will be at $\mathsf{S}_{l+m+n-2}[l+m-1]$ when $\mathsf{X}[l+m-2]$ is produced, and $\mathcal{F}_{ptw_m}$ will take the correct value with a probability of just $\frac{n-2}{n(n-1)}$.

In total, the probability that $\mathcal{F}_{ptw_m}$ takes the correct value is at least:

$$q_m \cdot \left(\frac{n-1}{n}\right)^{n-2} \cdot \frac{2}{n} + \left(1 - q_m \cdot \left(\frac{n-1}{n}\right)^{n-2}\right) \cdot \frac{n-2}{n(n-1)} \tag{7.28}$$

with

$$q_m = \left(\frac{n-1}{n}\right)^{m-1}\left(\frac{n-(m-1)}{n}\right) \cdot \prod_{a=1}^{m-1}\left(\frac{n-a}{n}\right) \tag{7.29}$$

7.3.4 Executing the attack

Executing the *PTW attack* is a little bit different than the previous attacks. First, as usual, we assume that an attacker has access to an Oracle $O_{SKIPWEP}(O_{WEP}(3,13,15),2)$ and starts collecting sessions. In all previous attacks, the attacker tried to determine $\mathsf{Rk}[0]$ first, before looking at $\mathsf{Rk}[1]$. Instead, the attacker now evaluates all functions $\mathcal{F}_{ptw_1}$ to $\mathcal{F}_{ptw_{13}}$ for every session. The result of each function is called a vote for σ_m having a specific value. The votes for each σ_m are stored in a separate table. We will call this table *frequency table* for σ_m.

Now, the attacker assumes that each top voted entry in each *frequency table* is the correct value for σ_m. These values are now tested for correctness. If they are, the attacker can simply calculate the key from these values.

Of course, this attack itself is expected to be weaker than the *Klein attack*, because every single function, except $\mathcal{F}_{ptw_1}$, which is the same as $\mathcal{F}_{Klein}$, has a lower success probability than $\mathcal{F}_{Klein}$. The only advantage of this attack so far is, that an implementation does not need to hold all sessions in the main memory, until the attack is finished. Instead, every session is not needed anymore, after all votes from that session have been added to the *frequency tables*.

The main advantage of the *PTW attack* is, that *key ranking* is much faster than in the *Klein attack*.

7.3.5 Key ranking with the PTW attack

Because all values for σ_m can be determined independently instead of sequentially than in all previous attacks, the key ranking strategies introduced in Section 7.2 can be modified to make use of this advantage. Of course, all these strategies can be used with the *PTW attack*, just by replacing a voting process with a lookup in the respective *frequency table*.

7.3.6 New strategies

The following strategies are now possible with the *PTW attack*:

Static number of choices for every frequency table This is just the same as *Static number of paths to follow*. Because the *PTW attack* can determine all values for σ_m independently of each other, we will describe this method again, without using the idea of a decision tree.

Let's assume that the attacker has processed all sessions and now has l_{key} frequency tables t_i. The attacker now assumes that the correct value for σ_m is in the k top voted entries in table t_m. The attacker can now start to test all possible values Rk can take, if the correct value for σ_m is in the k top voted entries in the frequency table t_m. There are at most $k^{l_{key}}$ possible values. Enumerating all these values usually takes less computational effort than testing them all.

Here, k is allowed to have a larger value then for a pre-PTW attack using the *Static number of paths to follow* key ranking strategies, because there are no new voting processes during the key ranking necessary. Additionally, it should be possible to offload this work to a FPGA, because just the top voted entries in every table t_m and a very small number of sessions (for example 5-10) are necessary to enumerate all possible keys and test them for correctness. The total size of the data structure can be less than 200 bytes for an efficient implementation.

Dynamic search borders In Section 7.2.1, we had the idea of creating a better key ranking strategy. A good strategy should try to try alternate decisions where the algorithm was very unsure about which decisions to take. For the *PTW attack*, this could be implemented as follows:

In the basic attack, the attacker assumes that for every m, the value σ_m was the most voted value in the respective voting table t_m. If there was a second value for σ_m, which had only a few less votes than the top voted entry in σ_m, the attacker was much more unsure about the decision for σ_m, than for an other value $\sigma_{m'}$, where the top voted entry has much more votes than the second most voted entry.

Let's call the number of possible values at the top of t_m, where possible values for σ_m are taken from, the *search border* for t_m. In our previous strategy, we assume that for every m, the correct value for σ_m is in the k top voted entries in t_m, and therefore, the search border for every frequency table t_m is k. Now we try a more dynamic approach. At the beginning, the search border for every table is 1, which means that just the top voted entry in the table is a possible candidate for σ_m. Now, we are looking for the table, where the first entry outside the search border has a minimal distance in number of votes to the top voted entry in the table. At this table, the search border is increased by one. Naturally speaking, we try to increase the search borders in tables, where we are unsure if our decision was correct. For tables where the top voted entry has much more votes than the next candidates, we where relatively sure about our decision and the search border is kept small.

This is iterated as long as a a certain value for the total number of keys is not exceeded. This strategy has additionally the advantage, that the number of possible keys can be fine controlled, because every increase of the search border will just increase the number of possible keys by factor 2 at most.

If the limit has been reached, all possible values for Rk are tested for their correctness. Let b_m be the search border for frequency table t_m. The total number of possible values for Rk is:

$$\prod_{a=0}^{l_{key}} b_a \tag{7.30}$$

We will later see that this approach has some additional advantages over just taking a static number of candidates for every frequency table, we do not know about yet.

8 Advanced versions of the PTW attack

So far, only the basic version of the PTW has been presented. There are a lot of possibilities, how the basic version of the attack can be improved. Again, all additions and subtractions in this Chapter, except for probabilities, are done mod n.

8.1 Brute forcing arbitrary key bytes

Some implementations of previous attacks do a brute force search on the last key byte, or even do a brute force search on the two last key bytes. The reason for this is, that the voting processes which are required to determine the last key bytes are more expensive in CPU-consumption than just testing all possible values for correctness.

Some implementations do a brute force search on the first key byte, because the number of votes for the first key byte is usually very low compared to the number of votes for other key bytes. This increases the required CPU-time up to factor 256 for the whole attack. In general, we can say, that a brute force search for other key bytes than the last ones are very expensive and perhaps even infeasible, when it comes to brute forcing a lot of key bytes. For every value tried at a key byte, all the following voting processes have to be repeated.

In the *PTW attack*, an attacker can do the voting process in the beginning and then do a brute force search on an arbitrary key byte, without heavily increasing the CPU-time of the attack. If no other key ranking method is used, this would only require to test 256 different keys, which would only require an unnoticeable small amount of CPU-time, compared to the voting process at the beginning. Even a brute force search on two or three key bytes seems to be possible without requiring much CPU-time. This can even be combined with a key ranking strategy, by just setting the search border of the frequency table t_m to the maximum, when $\mathsf{Rk}[m]$ should be brute forced.

At this moment, we may not see a reason for a brute force search. We will later see, that there are some circumstances, where an attacker wants to do a brute force search on some key bytes.

8.2 Correcting strong key bytes

In Section 7.3.3, we demanded that $\mathsf{S}_l[k] = \mathsf{S}_{l+m-1}[k] = \mathsf{S}_{l+m-1}[j_{l+m-1}]$ and assumed that this would happen with probability

$$\left(\frac{n-1}{n}\right)^{m-1} \left(\frac{n-(m-1)}{n}\right) \tag{8.1}$$

The left side is the probability that j takes the value k and the right side is the probability that i takes the value k. This is true for a random value k and the *generalized randomized RC4 stream cipher.* Unfortunately, there is a little problem with certain classes of keys.

The condition that j does not take the value k can be expressed as j_{l+m} does not take the value of a previous j after j_l. For some keys, j does usually take the value j_{l+m} after j_l and before j_{l+m}.

In an early stage of the RC4-KSA, the equation $\mathsf{S}[a] = a$ holds with a very high probability, because S is still very close to the initial permutation. This also means that most probably, $j_{l+m} = j_{l+m-1} + \mathsf{K}[l+m-1] + (l+m-1)]$ holds. If now $\mathsf{K}[l+m-1] = -(l+m-1)$ holds, j_{l+m-1} and j_{l+m} will most probably have the same value and $\mathcal{F}_{ptw_m}$ will just vote for a more or less random value.

In general, if a value r exists with $l+1 \leq r \leq l+m-1$ so that

$$\sum_{a=r}^{l+m-1} (\mathsf{K}[a] + a) = 0 \tag{8.2}$$

holds, the values j_r and j_{l+m} will most probably be equal, and the probability that $\mathcal{F}_{ptw_m}$ takes the correct value is not higher than $\frac{1}{n}$. Unfortunately, this depends on the value of our *root key* Rk. If there is a r with $1 \leq r \leq m-1$ so that

$$\sum_{a=r}^{m-1} (\mathsf{Rk}[a] + a + l_{iv}) = 0 \tag{8.3}$$

holds, the correct value for σ_{m-1} will just appear at a more or less random position in the frequency table and will most probably not be one of the top voted entries in the table. Therefore, the attack will most probably fail.

We call key byte $\mathsf{Rk}[m]$ a *strong key byte*, when it is resistant against the basic *PTW attack.* A *root key* is a *strong key*, if it has at least one *strong key byte.* A key byte which is not a *strong key byte* is a *weak key byte.*

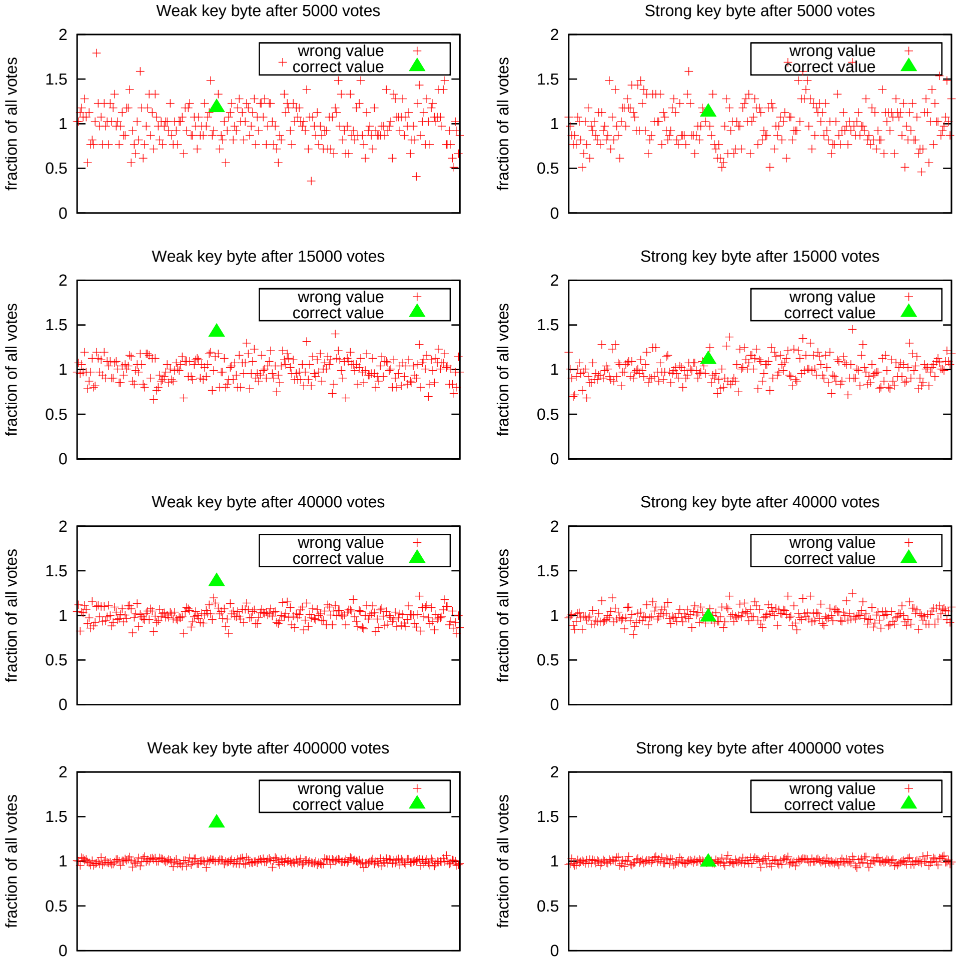


Figure 8.1: Votes for strong and non strong key bytes

Figure 8.1 shows the distribution of votes in the frequency table for σ_2 after 5000, 15000, 40000, and 400000 votes. As you can see, even after 400000 packets, all votes are nearly equally distributed for the strong key byte.

Fortunately, a good key ranking strategy can sometimes automatically fix this problem. Let's assume that an attacker has collected a very high number of sessions compared to the usual amount needed to successfully recover a *root key*. In all frequency tables, there will be one value which has received noticeable more votes than all other votes, which is the correct value, except for frequency tables for *strong key bytes*. Here, all values will share nearly a identical amount of votes as you can see in figure 8.1. An attacker who adjusts his search borders for key ranking dynamically, will perhaps only increase the border for frequency tables where the number of votes are nearly equal for all candidates and therefore consider all values in these tables as candidates. If the number of strong key bytes is low (for example 1 to 3), the total number of possible keys will still be small, so that they all can be tested using a reasonable amount of CPU-time. If the number of strong key bytes is higher, this approach will not be feasible and a better solution will be required.

One possibility would be to use the original *Klein attack* instead of the *PTW attack*, which has no problems with *strong key bytes*. We will later see in Section 8.4, that there are some scenarios, where the original *Klein attack* cannot be applied, but the *PTW attack* can be. So this is not always an option.

If we look at the condition for a *strong key byte* again, there are not a lot of possible values for r, at most 12 for the last key byte of a 13 byte root key Rk. Let's assume that an attacker knows the value for r and has already determined σ_0 to σ_{m-2} at this stage of the attack. The attacker can therefore calculate $\mathsf{Rk}[0]$ to $\mathsf{Rk}[m-2]$ from these values. By solving the equation for $\mathsf{Rk}[m-1]$, the attacker can now easily determine $\mathsf{Rk}[m-1]$ from $\mathsf{Rk}[0]$ to $\mathsf{Rk}[m-2]$ by the following formula:

$$\mathsf{Rk}[m-1] = (-m+1-l_{iv}) - \sum_{a=r}^{l+m-2} (\mathsf{Rk}[a] + a + l_{iv}) \tag{8.4}$$

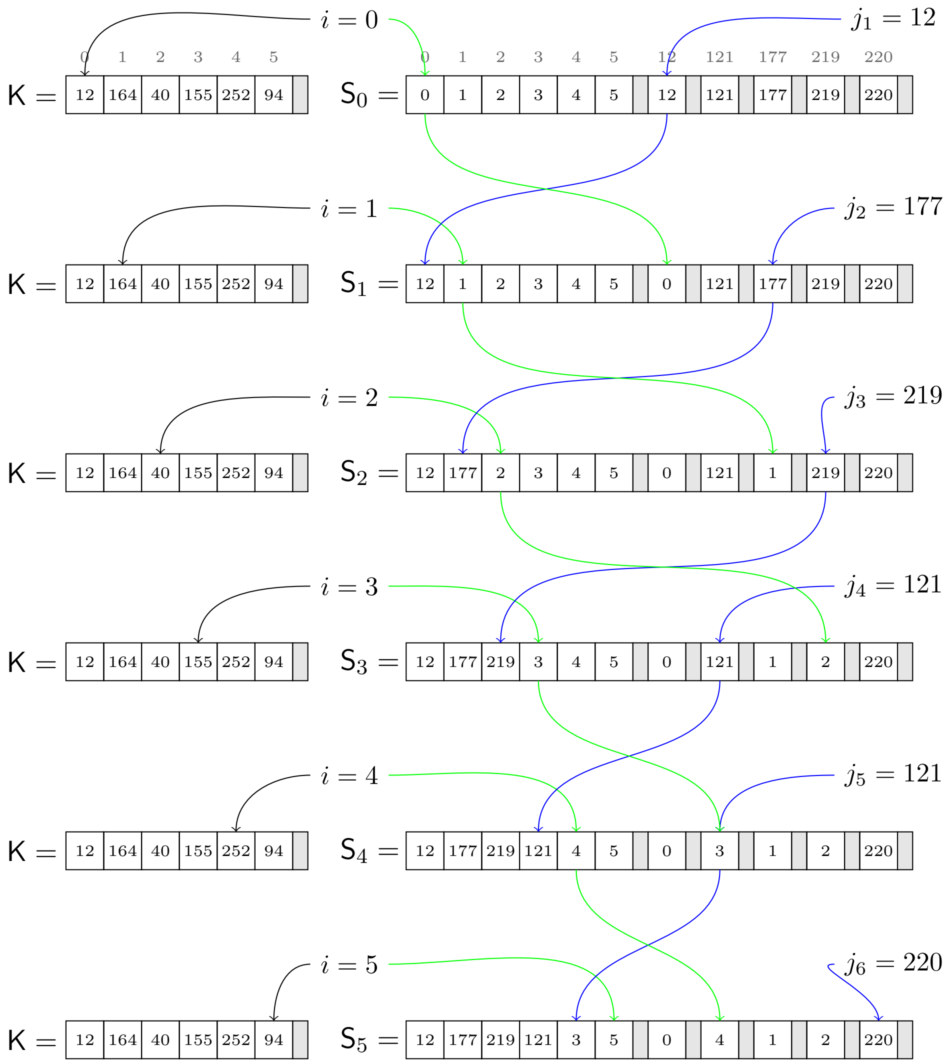
Determing which key bytes are *strong key bytes* can easily be done by looking at the distribution of the votes in each frequency table, if a high number of votes are available, but the attacker usually does not know r and has to try all possible values.

The attack can now be modified as follows. The attacker tries to identify which key bytes are strong key bytes using the distribution of votes in the respective frequency tables. For example the attacker could compute the χ^2 distance to a uniform distribution and the expected distribution for a non strong key byte, and for example assume that a table contained a strong key byte if the distance to the uniform distribution was lower than to the expected distribution for a non strong key byte. Another possibility would be to assume that there is just a single strong key byte, in the table having the lowest distance to a uniform distribution.

Let's assume that the attacker has decided that $\mathsf{Rk}[m]$ is a strong key byte. The attacker uses an arbitrary key ranking strategy and decides on the values for σ_0 to σ_{m-1} first, before trying to determine σ_m. When it comes to determine σ_m, the attacker tries all possible values for r and calculates σ_m using equation 8.4. Then the attack is continued with the remaining key bytes. Because r is at most 12 for a 13 byte *root key*, this approach is even feasible when the number of strong key bytes is high, because the total number of possible keys is still kept small, compared to trying all values in the *frequency table*. Another but less effective approach would be to do a brute force search on all frequency tables, whose key bytes are most probably strong. This requires much more CPU-time, but would be a little bit easier to implement.

8.2.1 An example

Let's assume that RC4 is used with the key $\mathsf{K} = 12, 164, 40, 155, 252, 94, 15, 163$. The attacker knows the first $l = 3$ bytes of the key and is interested in the sum $\mathsf{K}[3] + \mathsf{K}[4]$.



$i = 0$
$j_1 = 12$
0 1 2 3 4 5
0 1 2 3 4 5 12 121 177 219 220
$\mathsf{K} =$ 12 164 40 155 252 94
$\mathsf{S}_0 =$ 0 1 2 3 4 5 12 121 177 219 220
$i = 1$
$j_2 = 177$
$\mathsf{K} =$ 12 164 40 155 252 94
$\mathsf{S}_1 =$ 12 1 2 3 4 5 0 121 177 219 220
$i = 2$
$j_3 = 219$
$\mathsf{K} =$ 12 164 40 155 252 94
$\mathsf{S}_2 =$ 12 177 2 3 4 5 0 121 1 219 220
$i = 3$
$j_4 = 121$
$\mathsf{K} =$ 12 164 40 155 252 94
$\mathsf{S}_3 =$ 12 177 219 3 4 5 0 121 1 2 220
$i = 4$
$j_5 = 121$
$\mathsf{K} =$ 12 164 40 155 252 94
$\mathsf{S}_4 =$ 12 177 219 121 4 5 0 3 1 2 220
$i = 5$
$j_6 = 220$
$\mathsf{K} =$ 12 164 40 155 252 94
$\mathsf{S}_5 =$ 12 177 219 121 3 5 0 4 1 2 220

Figure 8.2: First 5 steps of RC4-KSA for $\mathsf{K} = 12, 164, 40, 155, 252, 94, 15, 163$

In the next step

$$\begin{aligned} j_4 &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] \\ &= 219 + 3 + 155 \\ &= 121 \end{aligned} \tag{8.5}$$

Now $\mathsf{S}_3[3] = 3$ and $\mathsf{S}_3[121] = 121$ are swapped. In the next step

$$\begin{aligned} j_5 &= j_4 + \mathsf{S}_4[4] + \mathsf{K}[4] \\ &= 121 + 4 + 252 \\ &= 121 \\ &= j_4 \end{aligned} \tag{8.6}$$

and $\mathsf{S}_4[4] = 4$ and $\mathsf{S}_4[121] = 3$ are swapped. Now, $\mathsf{S}_3^{-1}[3] = 3 \neq \mathsf{S}_4^{-1}[3]$ holds. For the rest of the RC4-KSA and the beginning of the RC4-PRGA, $\mathsf{S}_5[4]$ remains unchanged.

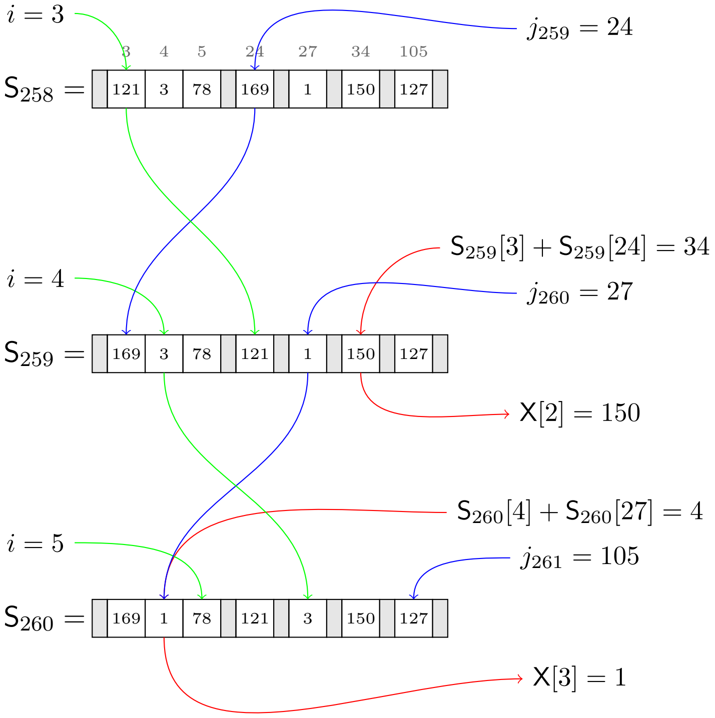


Figure 8.3: Generation of $\mathsf{X}[2]$ and $\mathsf{X}[3]$ for $\mathsf{K} = 12, 164, 40, 155, 252, 94, 15, 163$

When $\mathsf{X}[3]$ is produced, it correctly reveals $\mathsf{S}_{n+3}[4] = 3$. But an attacker who calculates:

$$\begin{aligned}
&\mathcal{F}_{ptw_2}(12, 164, 40, 1)\\
=&\mathsf{S}_3^{-1}[3 + 1 - \mathsf{X}[2]] - (j_3 + \mathsf{S}_3[3] + \mathsf{S}_3[3 + 1])\\
=&\mathsf{S}_3^{-1}[4 - 1] - (219 + 3 + 4)\\
=&\mathsf{S}_3^{-1}[3] - 226\\
=&3 - 226\\
=&33\\
\neq&\mathsf{S}_4^{-1}[3] - 226
\end{aligned} \tag{8.7}$$

would not have recovered $\mathsf{K}[3] + \mathsf{K}[4] = 151$.

An attacker who has guessed that $\mathsf{K}[4]$ is a strong key byte, and knows $\mathsf{K}[0]$ to $\mathsf{K}[2]$, knows that the only possible value for $\mathsf{K}[4]$ is 252 and would therefore have correctly recovered $\mathsf{K}[4]$.

8.3 Using more bytes of the key stream

If there are some of the following bytes of the key stream available, an attacker can make use of it. In RC4, the key is used cyclic in the RC4-KSA. Therefore, $\mathsf{K}||\mathsf{K}$ will result in the same internal state as K. Let's have a look at the function $\mathcal{F}_{ptw_{l_{key}+1}}$. This function is supposed [OFO+07, VV07] to take the value of $\sigma_{l_{key}} + \mathsf{IV}[0]$ with a higher probability than $\frac{1}{n}$. Because $\mathsf{IV}[0]$ is known by the attacker, he can subtract this value from the output of $\mathcal{F}_{ptw_{l_{key}+1}}$ and

$$\mathcal{F}_{ptw_{l_{key}+1}} - \mathsf{IV}[0] \tag{8.8}$$

get another vote for the sum of all key bytes of the *root key* $\sigma_{l_{key}-1}$.

Because the initialization vector has 3 bytes, the functions

$$\mathcal{F}_{ptw_{l_{key}+2}} - \mathsf{IV}[0] - \mathsf{IV}[1] \tag{8.9}$$

$$\mathcal{F}_{ptw_{l_{key}+3}} - \mathsf{IV}[0] - \mathsf{IV}[1] - \mathsf{IV}[2] \tag{8.10}$$

can be used too as votes for $\sigma_{l_{key}-1}$.

Because an attacker has now 4 times the votes for $\sigma_{l_{key}-1}$ than for any other value σ_m, he is usually able to determine the correct value for $\sigma_{l_{key}-1}$ with a much higher certainty than all other values σ_m.

After an attacker has determined $\sigma_{l_{key}-1}$, he can continue to gather additional votes for σ_0 to $\sigma_{l_{key}-2}$. The function

$$\mathcal{F}_{ptw_{l_{key}+l_{iv}+1+m}} - \mathsf{IV}[0] - \mathsf{IV}[1] - \mathsf{IV}[2] - \sigma_{l_{key}-1} \tag{8.11}$$

should take the value of σ_m with a higher probability than $\frac{1}{n}$ and the output can be used as an additional vote for the value of σ_m. An attacker might even use votes from the third or fourth repetition of the key in the RC4-KSA, but the success probability of $\mathcal{F}_{ptw_m}$ is close to $\frac{1}{n}$ for *root keys* with a length of 13 bytes. For 5 bytes long *root keys*, this approach might be more useful.

All votes for the value of σ_m are now combined in a single *frequency table* and the attacker can start to compute all remaining σ_m values (the last value $\sigma_{l_{key}-1}$ has already been determined) using an arbitrary key ranking strategy, except that the search border for $t_{l_{key}-1}$ is never increased. If the key is not found after having checked a reasonable amount of keys, the attacker might decide that his decision for $\sigma_{l_{key}-1}$ was incorrect and choose an alternative value for $\sigma_{l_{key}-1}$. The attacker now has to rectify all additional votes he computed under the assumption, that his first choice for $\sigma_{l_{key}-1}$ was correct. Rectification can be done efficiently by adding the old value for $\sigma_{l_{key}-1}$ to the vote and subtracting the new value for $\sigma_{l_{key}-1}$.

This method may be used to correct strong key bytes, because votes from a later step may still vote for the correct value σ_m with a higher probability than $\frac{1}{n}$, even if $\mathsf{Rk}[m]$ is a strong key byte.

8.4 Skipping some bytes of the key stream

Until now, we assumed that an attacker has access to sufficient number of key stream bytes as needed to perform the attack. For the original *FMS attack*, only the first byte of the key stream $\mathsf{X}[0]$ was needed. KoreK used the first two bytes of the key stream $\mathsf{X}[0]$ and $\mathsf{X}[1]$. For the *Klein attack* and the *PTW attack* $ks[l_{iv}-1]$ to $\mathsf{X}[l_{iv}+l_{key}-2]$ is needed to perform the attack. For a 13 bytes root key, this is $\mathsf{X}[2]$ to $\mathsf{X}[14]$ in a WEP scenario.

8.4.1 Key stream recovery

For a moment, we will leave the world of theoretical models and have a look at how an attacker can recover sessions in a real world scenario.

Custom packets

One of the easiest ways for an attacker to recover key streams is to use the *chopchop* or *fragmentation attack* to decrypt a single packet. The attacker can now use the key stream of this packet to generate a packet with arbitrary payload and inject this packet with a wireless client or the broadcast address as destination into the network again and again. When the packet is relayed by the access point, the packet is reencrypted. Because the attacker knows the plaintext of the packet, he can therefore recover the key stream which was used to encrypt the packet in full packet length.

This method has the disadvantage that the attacker must spend half of the bandwidth to inject packets and only the other half of the bandwidth is used to generate new sessions by the access point. We would like to have a faster method where a larger part of the bandwidth is used by fresh packets.

ARP injection

Let's assume that the network which is being attacked runs the IPv4 protocol. Today, this might be the case in more than 99% of all networks. I have personally never seen a wireless network which was not running IPv4 at all.

In IPv4, the ARP protocol [Plu82] is used to resolve IP-addresses to network addresses. If host A wants to send a packet to host B, but he does not know the hardware address of host B, he sends out an ARP request to the broadcast address, asking for the hardware address of host B. Host B will respond with an ARP respond telling A his hardware address. All hosts participating in IP communications keep track of the mapping from IP to hardware address in a so called *ARP table*.

Because the hardware address of a host might change from time to time (network card replaced, IP address moved to another machine) this is repeated from time to time. Another interesting feature of some wireless clients is, that they flush their *ARP table*, when they join a network. The original intention might be, that there might have been a different host in the last network the client was connected to, with the same address as a host in the new network, which is being joined now. If the client would still use the old hardware address, the host in the current network would not be reached. Because an attacker can force an arbitrary host to rejoin a network, this can be used to force an arbitrary host to flush his *ARP table*. The next packet send by this host triggers an ARP request, because the host now does not know the hardware address of any other host in the network.

Of course, these packets are all transmitted encrypted in a WEP network. To capture an *ARP request*, the attacker has to distinguish the encrypted request from all the other traffic on the network. Fortunately, the *ARP protocol* is quite simple. All *ARP packets* have an encrypted payload length of exactly 36 bytes, excluding the 4 byte CRC32 checksum. If the request is originating from a wired station, it is usually padded to the minimum size of an Ethernet frame and now has 54 bytes of payload length excluding the 4 byte CRC32 checksum.

The destination address of an *ARP request* is always the broadcast address of the network. Neither the destination address nor the exact packet length is hidden by WEP in any way. An attacker who is interested in an *ARP request* might just start looking for a packet with payload length 36 or 54 bytes and destination address *broadcast*. If all these conditions are met, he can assume that this packet is an *ARP request*. For example *aircrack-ng* uses this method to detect ARP requests.

Once an attacker captures an *ARP request*, he may start to reinject this request back into the network again and again. If both, the original sending station and the designated receiving station are both wireless clients, the injected request will generate 3 new packets:

1. The request will be relayed by the *access point* to the broadcast address of the network and the designated receiving station will receive the packet.
2. This station will generate an *ARP response* to that request and send the response to the *access point*.
3. The access point will relay that response to the original requesting station.

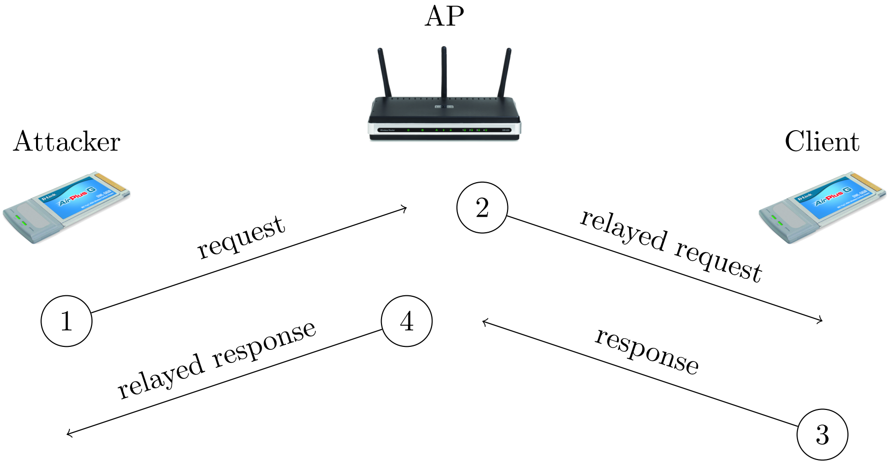


Figure 8.4: ARP injection

This approach has certain advantages:

- A lot of monitoring software like personal firewalls or perhaps IDS systems will just look for IP traffic and ignore ARP traffic. It is quite unlikely that such an attack is detected. Even if there is a monitoring system, these packets are valid packets in the network, just the packet rate will be higher than normal.

- ARP is usually not subject to rate limiting in most operating systems. Reaching a rate of more than 800 new packets per second is possible in an IEEE 802.11g network.
- The station who originated the first request will receive a lot of answers. All operating systems I have seen so far just drop these packets without displaying any kind of warning message.

Detecting *ARP response* packets in the traffic is a little bit harder, but possible too. An attacker starts again looking for packets with the specific length of 36 or 54 byte, not going to the destination address. Of course these packets could be any kind of short packets, but because the total fraction of *ARP responses* is very high during such an injection attack, the odds are high that these packets are really *ARP responses*.

The first 16 bytes of an *ARP request* or an *ARP response* are always a constant value, both type of packets only differ in the *ARP opcode* which is **00 01** for a request and **00 02** for a response. The next 6 bytes are the source address of the request or response. Because the source address is additionally transmitted in clear in a WEP network, an attacker can guess these values too. Figure 8.5 contains an illustration of the first bytes of an ARP packet. Fields marked with green have always a constant value or can easily be calculated from an encrypted packet. Fields marked with orange are sometimes known to an attacker, but are hard to be guessed by a fire-and-forget tool like *aircrack-ng*. Fields marked with red are usually not guessable by an attacker and must be assumed to be unknown.

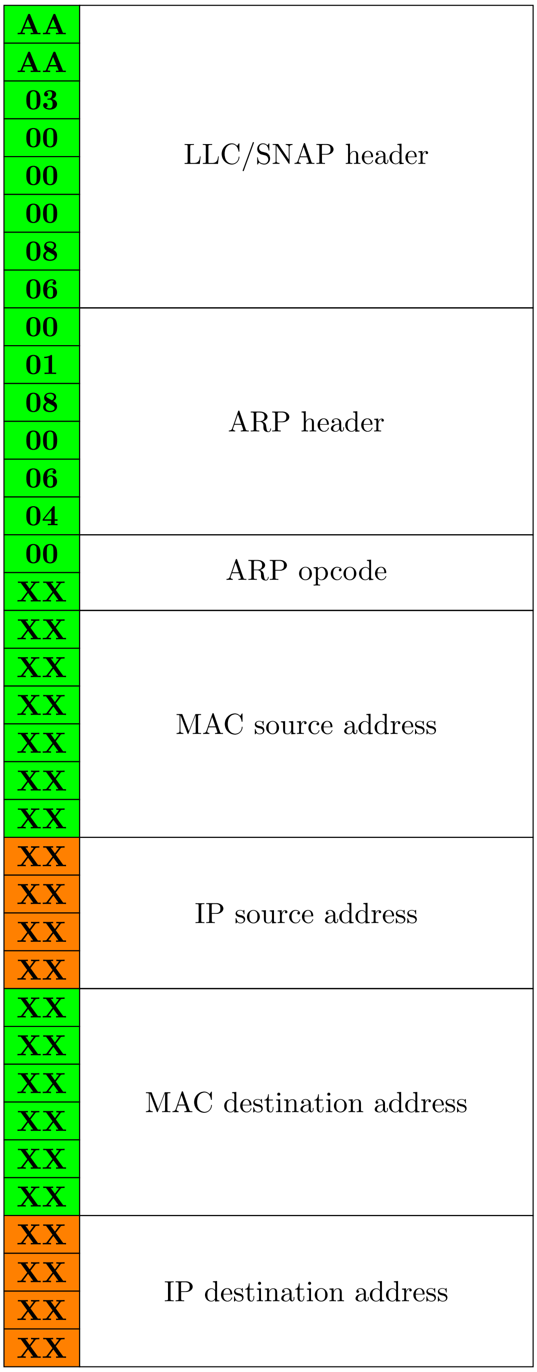
In total, the attacker can easily get the first 22 bytes of encrypted payload of every packet, by just using $\frac{1}{4}$ of the bandwidth for injecting packets. $\frac{3}{4}$ of the bandwidth can be used by fresh packets.

Of course, this attack is still an active attack which could be detected by a careful network operator. But we will see that we can even recover some key streams just by passively listen to a *WEP network*.

Passively listen to traffic

Let's assume that the target network is running the IPv4 protocol as before. Most of the packets in the network will be IPv4 packets. There might be some other packets like *ARP* or *Spanning Tree Protocol packets*, but these packets are only send seldom. If a packet has not the characteristics of an *ARP* or *Spanning Tree Protocol packet*, the attacker may assume that this is an IPv4 packet.

For an IPv4 packet, 11 out of the first 15 bytes of the encrypted plaintext can assumed to be constant values. 2 bytes, the so called *Fragmentation identification* is most times truly random and cannot be guessed by an attacker. One byte contains some flags and the high order bits of the fragmentation offset. All



AA	LLC/SNAP header
AA	
03	
00	
00	
00	
08	
06	
00	ARP header
01	
08	
00	
06	
04	
00	ARP opcode
XX	
XX	MAC source address
XX	
XX	
XX	
XX	
XX	
XX	IP source address
XX	
XX	
XX	
XX	MAC destination address
XX	
XX	
XX	
XX	
XX	
XX	IP destination address
XX	
XX	
XX	

Figure 8.5: ARP packet header

high order bits of the fragmentation offset can be assumed to be 0, and about 85% of all packets in a usual network have just the *don't fragment* flag set. All other packets have no flags set. The two bytes used to encode the length of the packet can be calculated, because WEP does not try to hide the exact length

🟩 AA	LLC/SNAP header
🟩 AA	
🟩 03	
🟩 00	
🟩 00	
🟩 00	
🟩 08	
🟩 00	
🟩 45	IP version and header length
🟩 00	Differentiated Services Field
🟩 XX	Total length
🟩 XX	
🟥 XX	Identification
🟥 XX	
🟩 40	Flags, msb. of frag. offset
🟩 00	lsb. of frag. offset
🟥 XX	Time to live
🟧 XX	Protocol
🟥 XX	Header checksum
🟥 XX	
🟧 XX	Source address
🟧 XX	
🟧 XX	
🟧 XX	
🟧 XX	Destination address
🟧 XX	
🟧 XX	
🟧 XX	

Figure 8.6: IPv4 packet header

of a data packet. In total, this means that an attacker always knows 13 bytes of the packet and can guess another byte X[14] correctly with a probability of about 85%. The two key stream bytes which contain the *fragmentation id* are X[12] and X[13]. Figure 8.6 contains an illustration of the first bytes of an IPv4 packet.

8.4.2 A passive PTW attack

An attacker who has mostly collected IPv4 packets will not therefore be able to find enough votes for the frequency tables t_{10} and t_{11}. For t_{12}, there are at least two possible approaches:

1. The attacker always assumes that the *don't fragment* flag was set, calculates the corresponding key stream and votes for a value σ_{13}.
2. The attacker uses a kind of partial vote. First, the attacker assumes that just *don't fragment* was set, calculates the key stream and calculates a vote for σ_{13}. Instead of a full vote, just a $\frac{85}{100}$ vote is added to the frequency table.

   Now, the attacker assumes that no flag was set, calculates a different key stream and calculates a different vote for σ_{13}. This vote is counted with $\frac{15}{100}$.

The attacker now sets the search border for t_{11} and t_{12} to the maximum and starts a *PTW attack* using an arbitrary key ranking method. Depending on the key ranking method, the attacker may decide to increase the search border for t_{13} first, because this frequency table contains the partial votes and the attacker was most uncertain about what to vote for here.

8.5 Using additional pre-PTW votes

Until now, we did only focus on the *Klein attack* and its multibyte extension. Clearly the *Klein attack* was the most effective attack we have seen so far, which iteratively computes the key. We will now see that it is possible to do a multibyte extension [VV07] of the *FMS attack*, which was the first key recovery attack against WEP.

The basic idea of the *FMS attack* can be seen as the following. If an attacker knows the first l words of a RC4 key, he can simulate the first l steps of the RC4-KSA. If the following conditions are holding after the first l steps of the RC4-KSA:

1. $\mathsf{S}_l[1] < l$
2. $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]] = l$

Then in the next step $\mathsf{S}_l[j_{l+1}]$ will be swapped to $\mathsf{S}_{l+1}[l]$. If none of the three values participate in any further swaps in the RC4-KSA, then the first word of output of the RC4-PRGA will be $\mathsf{S}_{l+1}[l]$ and therefore reveal j_{l+1} and the next key bytes $\mathsf{K}[l]$. The only exception would be, if the first word of output would be $\mathsf{S}_l[1]$ or $\mathsf{S}_l[\mathsf{S}_l[1]]$. This would indicate that $\mathsf{S}_l[1]$ or $\mathsf{S}_l[\mathsf{S}_l[1]]$ did participate in a further swap.

We can also rewrite these conditions as:

1. $\mathsf{S}_l[1] < l$
2. $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]] = k$

With $k = l$, this is the original *FMS attack*. The first condition ensures that $\mathsf{S}_l[1]$ is not changed by i in the remaining RC4-KSA. If the second condition is met, $\mathsf{S}_l[1] + \mathsf{S}_l[\mathsf{S}_l[1]]$ acts as a kind of *pointer* to $\mathsf{S}[k]$ so that hopefully, the first word of output of the RC4-PRGA will reveal the value of $\mathsf{S}_n[k]$. If $\mathsf{S}_{k+1}[k]$ did not participate in any further swaps after step k, and $\mathsf{S}_l[1]$ and $\mathsf{S}_l[\mathsf{S}_l[1]]$ did not participate in any further swaps after step l, this will reveal j_k. In total, we got:

$$\mathsf{X}[0] = \mathsf{S}_l[j_k] \tag{8.12}$$

or

$$\mathsf{S}_l^{-1}[\mathsf{X}[0]] = j_k \tag{8.13}$$

As in the PTW attack, we assume that

$$j_k = j_l + \sum_{a=l}^{k} (\mathsf{S}_l[a] + \mathsf{K}[a]) \tag{8.14}$$

holds. By solving this equation for $\sum_{a=l}^{k} \mathsf{K}[a]$, we get:

$$\mathcal{F}_{ptwfms_m}(\mathsf{K}[0], \ldots, \mathsf{K}[l-1], \mathsf{X}[0]) = \mathsf{S}_l^{-1}[\mathsf{X}[0]] - j_l - \sum_{a=l}^{l+m-1} \mathsf{S}_l[a] \tag{8.15}$$

with

$$\text{Prob}\left(\mathcal{F}_{ptwfms_m}(\mathsf{K}[0], \ldots, \mathsf{K}[l-1], \mathsf{X}[0]) = \sum_{a=l}^{l+m-1} \mathsf{K}[a]\right) > \frac{1}{n} \tag{8.16}$$

In general, rewriting correlations used by KoreK to multibyte correlations seems to be possible, but this is out of scope of this document.

8.5.1 An example

Let's assume that RC4 with the key $\mathsf{K} = 4, 255, 36, 127, 39, 185, 33, 206$ is used. The attacker knows the first $l = 3$ bytes of the key and is interested in $\mathsf{K}[3]+\mathsf{K}[4]$. After the first 3 steps of the RC4-KSA, $\mathsf{S}_3[0] = 4$ and $\mathsf{S}_3[1] = 0$. If $\mathsf{S}_3[0]$ and $\mathsf{S}_3[1]$ remain unchanged for the rest of the RC4-KSA, the first byte of output $\mathsf{X}[0]$ of the RC4-PRGA will be $\mathsf{S}[4]$. In the next step

$$\begin{aligned} j_4 &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] \\ &= 42 + 3 + 127 \\ &= 172 \end{aligned} \tag{8.17}$$

and $\mathsf{S}_3[3] = 3$ and $\mathsf{S}_3[172] = 172$ are swapped. In the next step

$$\begin{aligned} j_5 &= j_4 + \mathsf{S}_4[4] + \mathsf{K}[4] \\ &= 172 + 1 + 39 \\ &= 212 \\ &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] + \mathsf{S}_4[4] + \mathsf{K}[4] \\ &= j_3 + \mathsf{S}_3[3] + \mathsf{K}[3] + \mathsf{S}_3[4] + \mathsf{K}[4] \end{aligned} \tag{8.18}$$

and $\mathsf{S}_4[4] = 1$ and $\mathsf{S}_4[212] = 212$ are swapped. For the rest of the RC4-KSA $\mathsf{S}[0]$, $\mathsf{S}[1]$ and $\mathsf{S}[4]$ remain unchanged. This step is illustrated in figure 8.7.

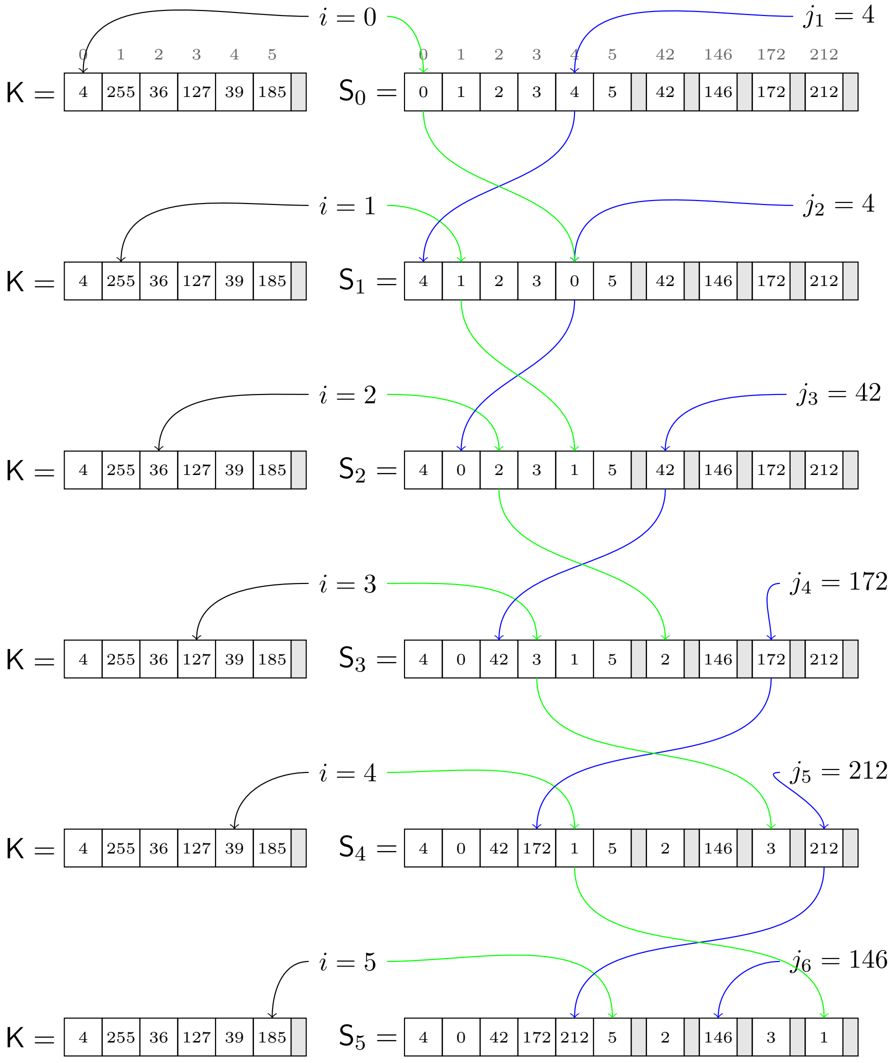

$i = 0$
$j_1 = 4$
0 1 2 3 4 5
0 1 2 3 4 5 42 146 172 212
$\mathsf{K} =$ 4 255 36 127 39 185
$\mathsf{S}_0 =$ 0 1 2 3 4 5 42 146 172 212
$i = 1$
$j_2 = 4$
$\mathsf{K} =$ 4 255 36 127 39 185
$\mathsf{S}_1 =$ 4 1 2 3 0 5 42 146 172 212
$i = 2$
$j_3 = 42$
$\mathsf{K} =$ 4 255 36 127 39 185
$\mathsf{S}_2 =$ 4 0 2 3 1 5 42 146 172 212
$i = 3$
$j_4 = 172$
$\mathsf{K} =$ 4 255 36 127 39 185
$\mathsf{S}_3 =$ 4 0 42 3 1 5 2 146 172 212
$i = 4$
$j_5 = 212$
$\mathsf{K} =$ 4 255 36 127 39 185
$\mathsf{S}_4 =$ 4 0 42 172 1 5 2 146 3 212
$i = 5$
$j_6 = 146$
$\mathsf{K} =$ 4 255 36 127 39 185
$\mathsf{S}_5 =$ 4 0 42 172 212 5 2 146 3 1

Figure 8.7: First 5 steps of RC4-KSA for $\mathsf{K} = 4, 255, 36, 127, 39, 185, 33, 206$

When the first byte of output by the RC4-PRGA is generated, $\mathsf{S}_n[0] = 4$ and $\mathsf{S}_n[1] = 0$ are swapped. The first byte of output is then $\mathsf{S}_{n+1}[\mathsf{S}_{n+1}[1]+\mathsf{S}_{n+1}[0]] = \mathsf{S}_{n+1}[4] = 212$. These step is illustrated in figure 8.8. An attacker who calculates

$$\begin{aligned}\mathcal{F}_{ptwfms_m}(4, 255, 36, 212) &= \mathsf{S}_3^{-1}[\mathsf{X}[0]] - j_3 - (\mathsf{S}_3[3] + \mathsf{S}_3[4]) \\ &= \mathsf{S}_3^{-1}[212] - 42 - (3 + 1) \\ &= 212 - 46 \\ &= 166 \\ &= 127 + 39 \\ &= \mathsf{K}[3] + \mathsf{K}[4]\end{aligned} \tag{8.19}$$

would have correctly recovered $\mathsf{K}[3] + \mathsf{K}[4]$.

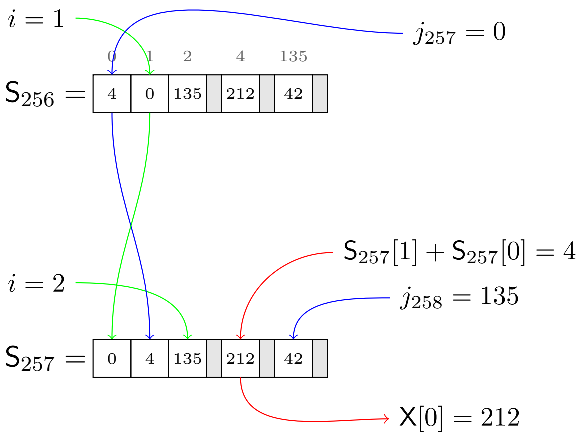


Figure 8.8: First byte of output for $\mathsf{K} = 4, 255, 36, 127, 39, 185, 33, 206$

8.6 Using some alternative correlations in RC4

Beside the KoreK attack and the Klein attack and their respective multibyte correlations, some other multibyte correlations where found in RC4, which have not yet been used for a real attack on RC4. For example *Subhamoy Maitra* and *Goutam Paul* found a multibyte correlation [MP07] which seems to be relatively independent from the Klein or FMS correlation.

8.7 Implementation

Aircrack-ng contains an implementation of the PTW attack, with some advanced features:

- It uses dynamic search borders for key ranking as described in section 7.3.6.
- It tries to correct up to two strong key bytes as described in section 8.2. It might automatically correct more strong key bytes, by using the dynamic search borders key ranking strategy. Additionally, the *Klein attack* is executed too, to recover keys with a very high number of strong key bytes.
- It can use passively captured IPv4 traffic too, as described in section 8.4.

To execute the PTW attack with all these advanced features on all packets captured in `/tmp/ptwtest.ivs`, an attacker has to execute the following command:

```
./aircrack-ng -0 /tmp/ptwtest.ivs
```

It is possible to disable the *Klein attack*, which is automatically executed to break keys with a lot of strong key bytes. The only reason to do this is for benchmarking the *PTW attack* only without the *Klein attack*. To execute only the *PTW attack* without the *Klein attack*, the following command has to be executed:

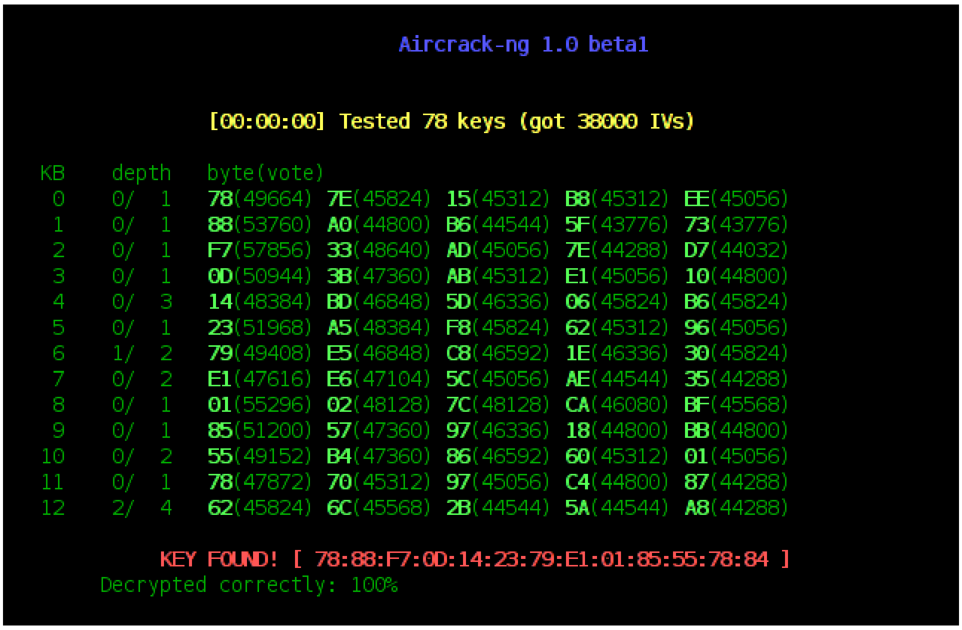
```
./aircrack-ng -P 1 -0 /tmp/ptwtest.ivs
```

If the attack was successful, an output similar to the one in figure 8.9 will be displayed.

8.8 Success rate

The success rate of the *PTW attack* as implemented in aircrack-ng is quite impressive. In total only 35,000 packets are needed to recover a secret *root key* with success probability 50% and 85% success probability can be reached using 45,000 packets. If a high number of packets are available, it is possible to recover more than 99.9% of all keys. Figure 8.10 contains all details.

If the *Klein attack* is disabled, the success rate is a little bit lower and just reaches about 99% for a lot of packets. The missing 1% of all keys are these keys with a lot of strong key bytes. Figure 8.11 contains the results.



```
                                 Aircrack-ng 1.0 beta1


                      [00:00:00] Tested 78 keys (got 38000 IVs)

   KB     depth    byte(vote)
    0     0/  1    78(49664) 7E(45824) 15(45312) B8(45312) EE(45056)
    1     0/  1    88(53760) A0(44800) B6(44544) 5F(43776) 73(43776)
    2     0/  1    F7(57856) 33(48640) AD(45056) 7E(44288) D7(44032)
    3     0/  1    0D(50944) 3B(47360) AB(45312) E1(45056) 10(44800)
    4     0/  3    14(48384) BD(46848) 5D(46336) 06(45824) B6(45824)
    5     0/  1    23(51968) A5(48384) F8(45824) 62(45312) 96(45056)
    6     1/  2    79(49408) E5(46848) C8(46592) 1E(46336) 30(45824)
    7     0/  2    E1(47616) E6(47104) 5C(45056) AE(44544) 35(44288)
    8     0/  1    01(55296) 02(48128) 7C(48128) CA(46080) BF(45568)
    9     0/  1    85(51200) 57(47360) 97(46336) 18(44800) BB(44800)
   10     0/  2    55(49152) B4(47360) 86(46592) 60(45312) 01(45056)
   11     0/  1    78(47872) 70(45312) 97(45056) C4(44800) 87(44288)
   12     2/  4    62(45824) 6C(45568) 2B(44544) 5A(44544) A8(44288)

               KEY FOUND! [ 78:88:F7:0D:14:23:79:E1:01:85:55:78:84 ]
          Decrypted correctly: 100%
```

Figure 8.9: aircrack-ng 1.0 beta 1 PTW results

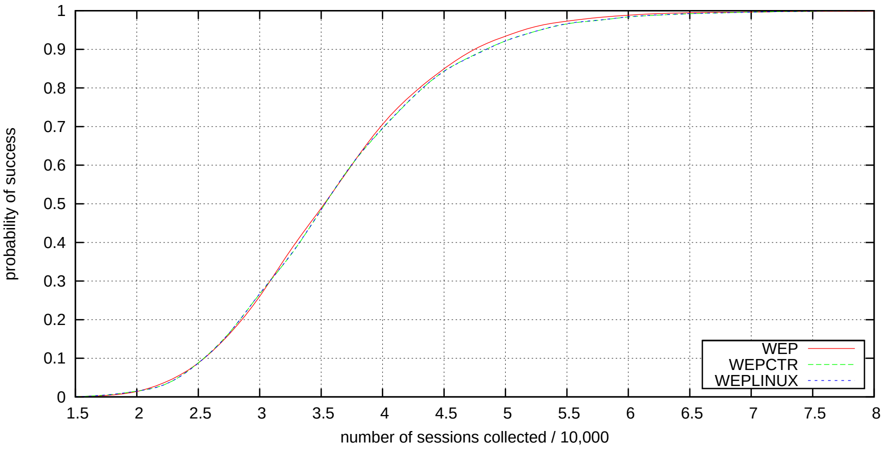


Figure 8.10: PTW success rate

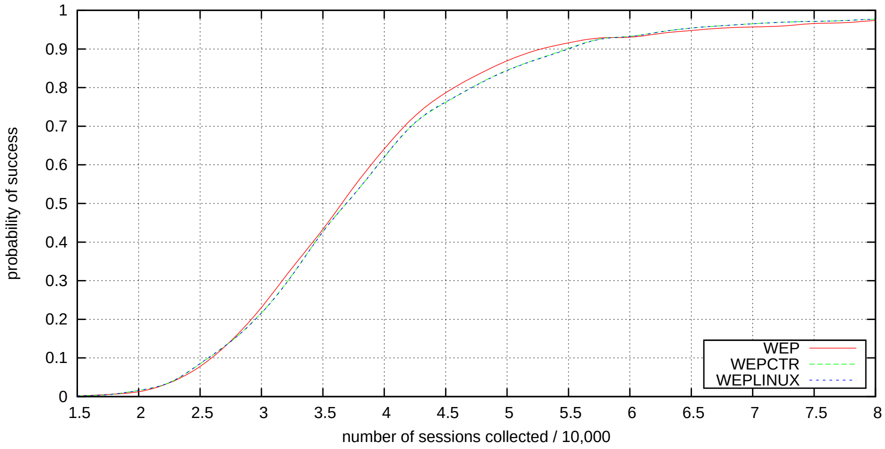


Figure 8.11: PTW without Klein success rate

9 WPA

When it became clear that WEP has some serious design problems, the *IEEE* started developing a successor protocol for WEP which was later named *Wi-Fi Protected Access* (WPA). WPA allows to modes how packets can be encrypted:

9.1 TKIP

The *Temporal Key Integrity Protocol* (TKIP) can be seen as a heavily modified version of WEP. To prevent attacks on the network, the following changes have been made to WEP:

1. A cryptographic message integrity code (MIC) is added to every packet before fragmentation. Unlike WEP, which encrypts and adds a CRC32 checksum to every fragment independently of each other, this prevents attacks like *fragmentation* or *chopchop*, where fragments of a packet are rearranged or packets are modified. It also protects the plaintext of the fragments to prevent an attacker from modifying the source or destination address of a packet.

   The designers of TKIP did not use an already existing algorithm like *SHA1HMAC* or *MD5HMAC* to calculate the MIC, because these algorithms need a lot of CPU time, and TKIP was designed to be usable on already existing hardware by installing a new firmware. Instead a new algorithm called *Michael* has been invented which is very fast compared to a *SHA1HMAC*.

2. To prevent attacks where an attacker tries to guess a checksum or attack the *Michael* algorithm with the help of a wireless station, TKIP only allows a small number of messages where the CRC32 checksum is correct, but the MIC is incorrect. Because CRC32 is still good in finding random transmission errors, such messages would indicate an attack. If more than two such messages are received by a station within a minute, TKIP is disabled for a minute and a renegotiation of the keys is suggested.

3. A per packet sequence counter (TSC) is used to prevent replay attacks. If a packet is received out of order, it is dropped by the receiving station. This prevents all kind of injection attacks, where a packet is replayed, like the ARP injection attack.

4. Unlike WEP, which changes only the first 3 bytes of the per packet key, which are the initialization vector, TKIP exchanges the per packet key completely after every single packet. The key mixing functions, which are used to generate the per packet keys, are designed to avoid values, which can be used for the *FMS attack*.

   All attacks on RC4 in this document are related key attacks, which require a high amount of related keys. By changing the complete key with a more or less random value, this seems to prevent all of these attacks.

Of course, TKIP still uses the RC4 stream cipher, but in a much more secure way than WEP does. The design goal of TKIP was not a protocol as secure as it could be. Instead a modification of WEP was created which prevents all known attacks, provides a quite high level of security and can be used on most existing hardware, just by installing a driver or firmware update.

9.2 AES CCMP

Alternatively, the *AES cipher* can be used in counter mode (CCMP) to encrypt the network traffic and to protect its integrity. This completely replaces the *RC4 stream cipher* and the *Michael algorithm* and provides a very high level of security. At this moment, there are no known realistic weaknesses in AES.

9.3 Key management

Additionally, an enterprise level *key management* was added to IEEE 802.11, which allows a lot of modes of authentication. Stations do not need to have a single secret pre-shared key anymore, instead a username and a password, smartcards, certificates, hardware security tokes and other authentication protocols can now be used. Except for broadcast traffic, every station uses individual keys to communicate with an access point, so that eavesdropping by another station in the same network is not possible anymore.

10 Conclusion

WEP has been known to be insecure before the PTW attack was published and has never been a real barrier for an attacker who was motivated to hack a network.

Before the *PTW attack* was published, an attacker had to collect at least 700,000 packets to execute the *KoreK attack* with a good success probability. At least 10 minutes where required to collect this amount of packets from a wireless network, sometimes more, if the signal quality is low. Not all attackers might be willing to spend this time to attack a network. For example, an attacker could be waiting in a restaurant, coffee bar or a railway station and be looking for cheap internet access. With the *PTW attack*, it is now possible to attack WEP protected networks in less than 60 seconds, so that an attacker does not even need a motivation or some free time to attack a network. The PTW attack can be fully automated so that a PDA with a wireless LAN card could be modified to automatically attack all networks it finds and recover their secret keys.

In 2004, *Andrea Bittau* published [BHL06] a tool called *wesside* which is a fully automatic WEP attack tool. *wesside* automatically scans for WEP protected networks and recovers a short key stream, by guessing the first bytes of an encrypted packet. This key stream is used to generate a longer key stream using the *fragmentation attack*. *wesside* trys to build a valid *ARP request* by decrypting some header fields from captured packets using the *fragmentation attack* or by just capturing an *ARP request*. This request is now injected into the network to generate network traffic, until enough packets have been captured to recover the secret key. This tool combined with a small PDA might be in the ultimate WEP attacking device, which can automatically collect secret WEP *root keys*, without any user interaction. Meanwhile, *wesside* has been integrated in the *aircrack-ng* toolsuite.

11 Acknowledgments and contributions

Most parts of this theses would not have been possible without the help of a lot of people. First, I would like to thank my two colleagues *Ralf-Philipp Weinmann* and *Andrei Pyshkin* who invented the PTW attack with me. Because our research is based on the *Klein attack*, it would not have been possible without the help of *Andreas Klein* who had the initial idea for these kinds of attacks.

Thanks go to all the other people of our working group, especially *Johannes Buchmann*, who allowed me to do this research as my diploma thesis. After we published our initial results, the public relationship department of our university and the *Chaos Computer Club* did coordinate most of the request from the media.

The *aircrack-ng team* did a great job with integrating our attack in the *aircrack-ng* toolsuite. With their help, we where even able to improve our attack and develop a ready to use passive version of our attack. *Andrea Bittau* gave me some good advices and integrated the *PTW attack* in the *wesside* tool, which can attack WEP protected networks fully automatically. *Martin Beck* contributed some patches to *aircrack-ng* to make it easier to benchmark. He also provided the results of the chopchop attack in Section 5.3.

This document has been proofread by *Nadja Kokic*, *Verena Moock*, and *Martin Beck*, who corrected a lot of errors and helped me to improve the readability.

Pictures of wireless LAN hardware have been provided by *Dlink*.

A Bibliography

[Arb01] William A. Arbaugh. An inductive chosen plaintext attack against wep/wep2. http://www.cs.umd.edu/~waa/attack/v3dcmnt.htm, 2001.

[BHL06] Andrea Bittau, Mark Handley, and Joshua Lackey. The final nail in WEP's coffin. In *IEEE Symposium on Security and Privacy*, pages 386–400. IEEE Computer Society, 2006.

[BO07] Shachar Bar-On. Hi-tech heist, how hi-tech thieves stole millions of customer financial records, Nov 2007. http://www.cbsnews.com/stories/2007/11/21/60minutes/main3530302.shtml.

[Cha06] Rafik Chaabouni. Break wep faster with statistical analysis. Technical report, EPFL, LASEC, June 2006.

[CWKS97] BP Crow, I. Widjaja, LG Kim, and PT Sakai. IEEE 802.11 Wireless Local Area Networks. *Communications Magazine, IEEE*, 35(9):116–126, 1997.

[Dör06] Stefan Dörhöfer. Empirische Untersuchungen zur WLAN-Sicherheit mittels Wardriving. Diplomarbeit, RWTH Aachen, September 2006. (in German).

[FMS01] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography 2001*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[Hul02] David Hulton. Practical exploitation of RC4 weakness in WEP environments, 2002. presented at HiverCon 2002.

[Jen96] Robert J. Jenkins. Isaac and rc4. [http://burtleburtle.net/bob/rand/isaac.html, 1996.

[Kle06] Andreas Klein. Attacks on the RC4 stream cipher. *submitted to Designs, Codes and Cryptography*, 2006.

[Kor04a] KoreK. chopchop (experimental WEP attacks). http://www.netstumbler.org/showthread.php?t=12489, 2004.

[Kor04b] KoreK. Next generation of WEP attacks? http://www.netstumbler.org/showpost.php?p=93942&postcount=35, 2004.

[Lab01] RSA Laboratories. Rsa security response to weaknesses in key scheduling algorithm of rc4. http://www.rsa.com/rsalabs/node.asp?id=2009, 2001.

[Man05] Itsik Mantin. A practical attack on the fixed rc4 in the wep mode. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 395–411. Springer, 2005.

[Mat94] M. Matsui. The First Experimental Cryptanalysis of the Data Encryption Standard. *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–11, 1994.

[Men01] A.J. Menezes. *Handbook of Applied Cryptography*. CRC Press, 2001.

[Mir02] Ilya Mironov. (not so) random shuffles of rc4. In *Advances in Cryptology - CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18-22, 2002. Proceedings*, volume 2442, pages 304–319, 2002.

[MP07] Subhamoy Maitra and Goutam Paul. New form of permutation bias and secret key leakage in keystream bytes of rc4. Cryptology ePrint Archive, Report 2007/261, 2007. http://eprint.iacr.org/.

[MS02] I. Mantin and A. Shamir. A Practical Attack on Broadcast RC4. *Fast Software Encryption: 8th International Workshop, FSE 2001, Yokohama, Japan, April 2-4, 2001: Revised Papers*, 2002.

[OFO+07] Yuko Ozasa, Yoshiaki Fujikawa, Toshihiro Ohigashi, Hidenori Kuwakado, and Masakatu Morii. A study on the Tews, Weinmann, Pyshkin attack against WEP. In *IEICE Tech. Rep.*, volume 107 of *ISEC2007-47*, pages 17–21, Hokkaido, July 2007. Thu, Jul 19, 2007 - Fri, Jul 20 : Future University-Hakodate (ISEC, SITE, IPSJ-CSEC).

[Plu82] D. C. Plummer. RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware, November 1982.

[Riv92] RL Rivest. The RC4 Encryption Algorithm. RSA Data Security. *Inc., March*, 12, 1992.

[Sch00] W. Schindler. A Timing Attack against RSA with the Chinese Remainder Theorem. *CHES 2000*, pages 109–124, 2000.

[SIR04] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *ACM Transactions on Information and System Security*, 7(2):319–332, May 2004.

[Ste94] David Sterndark. Rc4 algorithm revealed. Usenet posting, Message-ID: <sternCvKL4B.Hyy@netcom.com>, Sep 1994.

[TWP07] Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin. Breaking 104 bit wep in less than 60 seconds. Cryptology ePrint Archive, Report 2007/120, 2007. http://eprint.iacr.org/.

[VV07] Serge Vaudenay and Martin Vuagnoux. Passive-only key recovery attacks on RC4. In *Selected Areas in Cryptography 2007*, Lecture Notes in Computer Science. Springer, 2007. to appear.

B List of Figures

C Index

D Notes