# Breaking 104 bit WEP in less than 60 seconds

Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin
<e_tews,weinmann,pyshkin@cdc.informatik.tu-darmstadt.de>

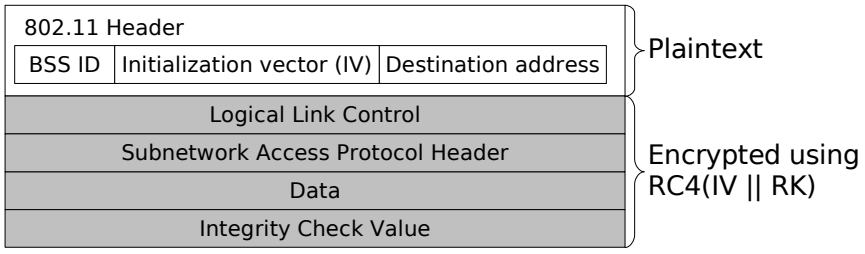
Technische Universität Darmstadt,
Fachbereich Informatik
Hochschulstrasse 10
D-64289 Darmstadt

**Abstract.** We demonstrate an active attack on the WEP protocol that is able to recover a 104-bit WEP key using less than 40.000 frames with a success probability of 50%. In order to succeed in 95% of all cases, 85.000 packets are needed. The IV of these packets can be randomly chosen. This is an improvement in the number of required frames by more than an order of magnitude over the best known key-recovery attacks for WEP. On a IEEE 802.11g network, the number of frames required can be obtained by re-injection in less than a minute. The required computational effort is approximately $2^{20}$ RC4 key setups, which on current desktop and laptop CPUs is neglegible.

## 1 Introduction

Wired Equivalent Privacy (WEP) is a protocol for encrypting wirelessly transmitted packets on IEEE 802.11 networks. Although it is known to be insecure and has been superseded by Wi-Fi Protected Access (WPA) [8], it still is in widespread use. In a WEP-protected network, radio stations share a common key, the *root key* $\mathsf{Rk}$. A successful recovery of this key gives an attacker full access to the network. The packet data transmitted is protected by an ICV (Integrity Check Value) computed to protect against manipulation[1]. For each packet, a 24-bit $\mathsf{IV}$ (initialization vector) is chosen. A per packet key $\mathsf{K} = \mathsf{IV}||\mathsf{Rk}$ key is used to encrypt the packet and the ICV using the RC4 stream cipher. This encrypted data then is transmitted prepended with the IV. Figure 1 shows a 802.11 frame in detail.

**Fig. 1.** A 802.11 frame encrypted using WEP




The packet length as well as the link layer source and destination addresses are not masked by the protocol. The protocol defines no measures against replay attacks.

[1] the ICV being a CRC32 checksum, it does not provide integrity in the cryptographical sense. The ICV rather is another layer of protection against accidental data corruption.

RC4 is a widely used stream cipher that was invented by Ron Rivest of RSA Security in 1987. It was a trade secret until 1994 when the algorithm was published anonymously on the Internet. RC4's internal internal state consists of a 256-byte array $\mathsf{S}$ that defines a permutation as well as two integers $0 \leq i, j \leq 255$ acting as pointers into the array.

The RC4 key setup initializes the internal state using a key $\mathsf{K}$ of arbitrary length up to 256 bytes. By exchanging elements of the state in each step, it incrementally transforms the identity permutation into a "random" permutation. The quality of randomness of the permutation after the key setup will be further analyzed in Section 3.

The RC4 key stream generation algorithm updates the RC4 internal state and generates one byte of key stream. The key stream is XORed to the plaintext to generate the ciphertext.

In 2001, Fluhrer, Mantin and Shamir presented a related-key ciphertext-only attack against RC4 [3]. In order for this attack to work, the IVs need to fulfill a so-called "resolved condition". This attack was suspected to be applicable to WEP, which was later demonstrated by Stubblefield et. al [7]. The number of frames with different IVs that are needed to fully determine the complete key in this attack scenario is estimated to be approximately 4 million.

More recently, Klein [5] showed an improved way of attacking RC4 that does not need the "resolved condition" on the IVs and gets by with a significantly reduced number of frames.

**Listing 1.1.** RC4 key setup

```
1  for i ← 0 to 255 do
2    S[i] ← i
3  end
4  j ← 0
5  for i ← 0 to 255 do
6    j ← j+S[i]+K[i mod len(K)] mod 256
7    swap(S, i, j)
8  end
9  i ← 0
10 j ← 0
```

**Listing 1.2.** RC4 key stream generation

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

Table 1 shows a statistic of employed encryption methods in a sample of 490 networks, found somewhere in the middle of Germany in March 2007. Another survey of more than 15.000 networks was performed in a larger German city in September 2006 [2]. Both data sets demonstrate that WEP is still the most popular method for securing wireless networks. Similar observations have been made by Bittau, Handley and Lackey [1]. Their article also give an excellent history of WEP attacks and describes a real-time decryption attack based on packet fragmentation that does not recover the key.

The structure of the paper is as follows: In Section 2 we introduce the notation that is used throughout the rest of this paper, in Section 3 we present a summary of Klein's attack on RC4, in Section 4 we specialize Klein's Attack to WEP, Section 5 describes how sufficient amounts of key stream can be obtained for the attack, Section 6 describes extensions of the attack such as key ranking techniques in detail and Section 7 gives experimental results.

**Table 1.** Methods used for securing wireless networks

| Time | No Encryption | WEP | WPA1 | WPA2 |
|---|---|---|---|---|
| March 2007 | 21.8% | 46.3% | 19.6% | 7.3% |
| Middle of 2006 | 23.3% | 59.4% | 14.5% | 3.3% |

## 2 Notation

For arrays or vectors we use the $[\cdot]$ notation, as used in many programing languages like in *C* or *Java*. All indices start at 0. For permutations $\mathsf{P}$ we use $\mathsf{P}^{-1}$ for the inverse permutation meaning $\mathsf{P}[i] = j \Leftrightarrow \mathsf{P}^{-1}[j] = i$. We will use $x \approx_n y$ as a short form for $x \approx y \bmod n$.

We have a closer look at the RC4 key setup algorithm described in listing 1.1, especially at the values for $\mathsf{S}$, $i$ and $j$. After line 4, $\mathsf{S}$ is the identity permutation and $j$ has the value 0. We will use $\mathsf{S}_k$ and $j_k$ for the values of $\mathsf{S}$ and $j$ after $k$ loops starting in line 5 have been completed. For example, if the key `CA FE BA BE` is used, $\mathsf{S}_0$ is the identity permutation and $j_0 = 0$. After the first key byte has been processed, $j_1 = 202$ and $\mathsf{S}_1[0] = 202$, $\mathsf{S}_1[202] = 0$, and $\mathsf{S}_1[x] = \mathsf{S}_0[x] = x$ for $0 \neq x \neq 202$.

$\mathsf{Rk}$ is the WEP or root key and $\mathsf{IV}$ is the initialization vector for a packet. $\mathsf{K} = \mathsf{Rk}||\mathsf{IV}$ is the session or per packet key. $\mathsf{X}$ is a key stream generated using $\mathsf{K}$. We will refer to a key stream $\mathsf{X}$ with the corresponding initialization vector $\mathsf{IV}$ as a session.

## 3 Klein's attack on RC4

Suppose $w$ key streams were generated by RC4 using packet keys with a fixed root key and different initialization vectors. Denote by $\mathsf{K}_u = (\mathsf{K}_u[0], \ldots, \mathsf{K}_u[m]) = (\mathsf{IV}_u||\mathsf{Rk})$ the $u$th packet key and by $\mathsf{X}_u = (\mathsf{X}_u[0], \ldots, \mathsf{X}_u[m-1])$ the first $m$ bytes of the $u$th key stream, where $1 \leq u \leq w$. Assume that an attacker knows the pairs $(\mathsf{IV}_u, \mathsf{X}_u)$ – we shall refer to them as *samples* – and tries to find $\mathsf{Rk}$.

In [5], Klein showed that there is a map $\mathcal{F}_i \colon (\mathbb{Z}/n\mathbb{Z})^i \to \mathbb{Z}/n\mathbb{Z}$ with $1 \leq i \leq m$ such that

$$\mathcal{F}_i(K[0], \ldots, \mathsf{K}[i-1], \mathsf{X}[i-1]) = \begin{cases} \mathsf{K}[i], & \text{with Prob} \approx \frac{1.36}{n} \\ a \neq \mathsf{K}[i], & \text{with Prob} < \frac{1}{n} \text{ for all } a \end{cases}$$

If the first $i$ bytes of a packet key are known, then the internal permutation $S_{i-1}$ and the index $j$ at the $(i-1)$th step of the RC4 key setup algorithm can be found. We have

$$\mathcal{F}_i(\mathsf{K}[0], \ldots, \mathsf{K}[i-1], \mathsf{X}[i-1]) = \mathsf{S}_{i-1}^{-1}[i - \mathsf{X}[i-1]] - (j_{i-1} + \mathsf{S}_{i-1}[i]) \bmod n$$

The attack is based on the following properties of permutations.

**Theorem 1** *For a random permutation* $\mathsf{P}$*, and random number* $j \in \{0, \ldots, n-1\}$*, we have*

$$\text{Prob}(\mathsf{P}[j] + \mathsf{P}[\mathsf{P}[i] + \mathsf{P}[j] \bmod n] = i \bmod n) = \frac{2}{n}$$
$$\text{Prob}(\mathsf{P}[j] + \mathsf{P}[\mathsf{P}[i] + \mathsf{P}[j] \bmod n] = c \bmod n) = \frac{n-2}{n(n-1)}$$

*where* $i, c \in \{0, \ldots, n-1\}$ *are fixed, and* $c \neq i$*.*

*Proof.* see [5].

In the case of $n = 256$, the first probability is equal to $2^{-7} \approx 0.00781$, and the second one is approximately equal to 0.00389.

From Theorem 1 it follows that for RC4 there is a correlation between $i$, $\mathsf{S}_{i+n}[\mathsf{S}_{i+n}[i] + \mathsf{S}_{i+n}[j] \bmod n]$, and $\mathsf{S}_{i+n}[j] = \mathsf{S}_{i+n-1}[i]$.

Next, the equality $\mathsf{S}_i[i] = \mathsf{S}_{i+n-1}[i]$ holds with high probability. The theoretical explanation of this is the following. If we replace the line 6 of the RC4 key setup, and the line 2 of the RC4 key stream generator by $j \leftarrow \mathrm{RND}(n)$, [2] then

$$\mathrm{Prob}(\mathsf{S}_i[i] = \mathsf{S}_{i+n-1}[i]) = \left(1 - \frac{1}{n}\right)^{n-2} \approx e^{-1}$$

Moreover, we have $\mathsf{S}_i[i] = \mathsf{S}_{i-1}[j_i] = \mathsf{S}_{i-1}[j_{i-1} + \mathsf{S}_{i-1}[i] + \mathsf{K}[i] \bmod n]$.

Combining this with Theorem 1, we get the probability that $\mathsf{K}[i] = \mathsf{S}_{i-1}^{-1}[i - \mathsf{S}_{i+n-1}[\mathsf{S}_{i+n-1}[i] + \mathsf{S}_{i+n-1}[j] \bmod n] \bmod n] - (j_{i-1} + \mathsf{S}_{i-1}[i])$ is approximately equal to

$$\left(1 - \frac{1}{n}\right)^{n-2} \frac{2}{n} + \left(1 - \left(1 - \frac{1}{n}\right)^{n-2}\right) \frac{n-2}{n(n-1)} \approx \frac{1.36}{n}$$

## 4 Extension to multiple key bytes

With Klein's attack, it is possible to iteratively compute all secret key bytes if enough samples are available. This iterative approach has a significant disadvantage: In this case the key streams and IVs need to be saved and processed for every key byte. Additionally correcting falsely guessed key byte is expensive, because the computations for all key bytes following $\mathsf{K}[i]$ needs to be repeated if $\mathsf{K}[i]$ was incorrect.

We extend the attack such that is it possible to compute key bytes independently of each other and thus make efficient use of the attack possible by using key ranking techniques. Klein's attack is based on the the fact that

$$\mathsf{K}[i] \approx_n \mathsf{S}_i^{-1}[i - \mathsf{X}[i-1]] - (\mathsf{S}_i[i] + j_i) \tag{1}$$
$$\mathsf{K}[i+1] \approx_n \mathsf{S}_{i+1}^{-1}[(i+1) - \mathsf{X}[(i+1)-1]] - (\mathsf{S}_{i+1}[i+1] + j_{i+1}) \tag{2}$$

We may write $j_{i+1}$ as $j_i + \mathsf{S}_i[i] + \mathsf{K}[i]$. By replacing $j_{i+1}$ in equation 2, we get an approximation for $\mathsf{K}[i] + \mathsf{K}[i+1]$:

$$\mathsf{K}[i+1] \approx_n \mathsf{S}_{i+1}^{-1}[(i+1) - \mathsf{X}[(i+1)-1]] - (\mathsf{S}_{i+1}[i+1] + j_i + \mathsf{S}_i[i] + \mathsf{K}[i]) \tag{3}$$
$$\mathsf{K}[i] + \mathsf{K}[i+1] \approx_n \mathsf{S}_{i+1}^{-1}[(i+1) - \mathsf{X}[(i+1)-1]] - (\mathsf{S}_{i+1}[i+1] + j_i + \mathsf{S}_i[i]) \tag{4}$$

By repeatedly replacing $j_{i+k}$, we get an approximation for $\sum_{l=i}^{i+k} \mathsf{K}[l]$. Because we are mostly interested in $\sum_{l=3}^{3+i} \mathsf{K}[l] = \sum_{l=0}^{i} \mathsf{Rk}[l]$ in a WEP scenario, we will use the symbol $\sigma_i$ for this sum.

[2] Some publications approximate $\left(1 - \frac{1}{n}\right)^{n-2}$ by $\frac{1}{e}$. We will use $\left(1 - \frac{1}{n}\right)^{n-2}$ for the rest of this paper.

$$\sigma_i \approx_n \mathsf{S}_{3+i}^{-1}[(3+i) - \mathsf{X}[2+i]] - \left(j_3 + \sum_{l=3}^{i+3} \mathsf{S}_l[l]\right) = \tilde{\mathcal{A}}_i \tag{5}$$

The right side of equation 5 still depends on the key bytes $\mathsf{K}[3]$ to $\mathsf{K}[i-1]$, because they are needed to compute $\mathsf{S}_l$ and $\mathsf{S}_i^{-1}$. By replacing them with $\mathsf{S}_3$, we get another approximation $\mathcal{A}_i$ for $\sigma_i$, which only depends on $\mathsf{K}[0]$ to $\mathsf{K}[2]$.

$$\sigma_i \approx_n \mathsf{S}_3^{-1}[(3+i) - \mathsf{X}[2+i]] - \left(j_3 + \sum_{l=3}^{i+3} \mathsf{S}_3[l]\right) = \mathcal{A}_i \tag{6}$$

Under idealized conditions, Klein derives the following probability for the event $\tilde{A}_i = \sigma_i$:

$$\text{Prob}\left(\sigma_i = \tilde{\mathcal{A}}_i\right) \approx \left(1 - \frac{1}{n}\right)^{n-2} \cdot \frac{2}{n} + \left(1 - \left(1 - \frac{1}{n}\right)^{n-2}\right) \cdot \frac{n-2}{n(n-1)} \tag{7}$$

The first part of sum represents the probability that $\mathsf{S}[i+3]$ remains unchanged until $\mathsf{X}[2+i]$ is generated, the second part represents the probability that $\mathsf{S}[i+3]$ is changed during key scheduling or key stream generation with $A_i$ still taking the correct value. By replacing $\mathsf{S}_l$ and $\mathsf{S}_{i+3}$ with their previous values, we have reduced that probability slightly.

$\mathsf{S}_{k+3}[k+3]$ differs from $\mathsf{S}_3[k+3]$ only if one of the values of $j_3$ to $j_{k+2}$ has been $k+3$. All values of $\mathsf{S}_l[l]$ will be correct, if for all $j_z$ with $3 \leq z \leq 3+i$ the condition $j_z \notin \{z, \ldots, 3+i\}$ holds. Assuming $j$ changes randomly, this happens with probability $\prod_{k=1}^{i}\left(1 - \frac{k}{n}\right)$. Additionaly $\mathsf{S}_{3+i}[j_{i+3}]$ should not be changed between iteration 3 and $3+i$. This is true if $j$ does not take the value of $j_{i+3}$ in a previous round, which happens with probability $\approx \left(1 - \frac{1}{n}\right)^i$ and $i$ does not take the value of $j_{i+3}$, which happens with probability $\approx \left(1 - \frac{i}{n}\right)$. To summarize, the probability that replacing all occurrences of $\mathsf{S}$ in $\tilde{\mathcal{A}}_i$ with $\mathsf{S}_3$ did not change anything is:

$$q_i = \left(1 - \frac{1}{n}\right)^i \cdot \left(1 - \frac{i}{n}\right) \cdot \prod_{k=1}^{i}\left(1 - \frac{k}{n}\right) \tag{8}$$

This results in the following probability of $\mathcal{A}_i$ taking the correct value for $\sigma_i$.

$$\text{Prob}\left(\sigma_i = \mathcal{A}_i\right) \approx q_i \cdot \left(1 - \frac{1}{n}\right)^{n-2} \cdot \frac{2}{n} + \left(1 - q_i \cdot \left(1 - \frac{1}{n}\right)^{n-2}\right) \cdot \frac{n-2}{n(n-1)} = p_{\text{correct}_i} \tag{9}$$

Experimental results using more than *50,000,000,000* simulations with 104 bit WEP keys show, that this approximations differs less than 0.2% from values determined from these simulations.

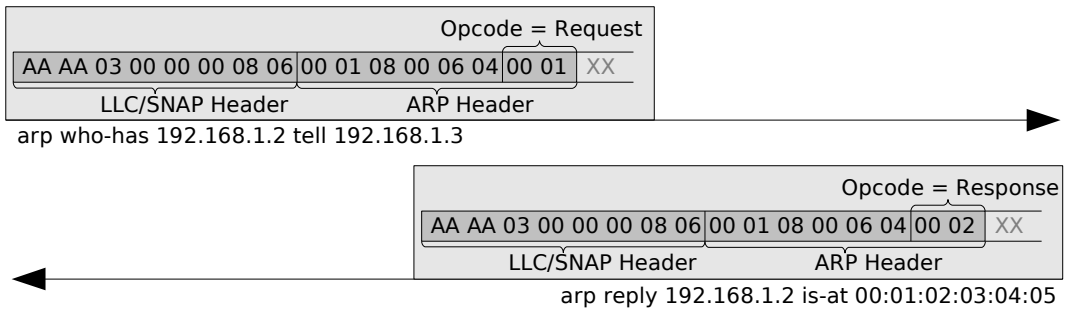
## 5 Obtaining sufficient amounts of key stream

The Internet Protocol (IP) is the most widely deployed network protocol. For our attack to work, we assume that version 4 (IPv4) of this protocol is used on the wireless networks we attack.

If host A wants to send an IP datagram to host B, A needs the physical address of host B or the gateway through which B can be reached. To resolve IP addresses of hosts to their physical address, the Address Resolution Protocol (ARP) [6] is used. This works as follows: Host A sends an ARP request to the link layer broadcast address. This request announces that A is looking for the physical address of host B. Host B responds with an ARP reply containing his own physical address to host A. Since the Address Resolution Protocol a link layer protocol it is typically not restricted by any kind of packet filters or rate limiting rules.

ARP requests and ARP replies are of fixed size. Because the size of a packet is not masked by WEP, they can usually be easily distinguished from other traffic. The first 16 bytes of cleartext of an ARP packet are made up of a 8 byte long 802.11 Logical Link Control (LLC) header followed by the first 8 bytes of the ARP packet itself. The LLC header is fixed for every ARP packet (`AA AA 03 00 00 00 08 06`). The first 8 bytes of an ARP request are also fixed. Their value is `00 01 08 00 06 04 00 01`. For an ARP response, the last byte changes to `02`, the rest of the bytes are identical to an ARP request. An ARP request is always sent to the broadcast address, while an ARP response is sent to a unicast address. Because the physical addresses are not encrypted by WEP, it is easy to distinguish between an encrypted ARP request and response.

**Fig. 2.** Cleartext of ARP request and response packets




By XORing a captured ARP packet with these fixed patterns, we can recover the first 16 bytes of the key stream. The corresponding IV is transmitted in clear with the packet.

To speed up key stream recovery, it is possible to re-inject a sniffed ARP request again into the network. The destination answers the request with a new response packet that we can add to our list of key streams. If the initiator and the destination of the original request have been both wireless stations, every reinjected packet will generate 3 new packets, because the transmission will be relayed by the access point. Because ARP replies expire quickly, it usually takes only a few seconds or minutes until an attacker can capture an ARP request and start reinjecting it. The first public implementation of a practical re-injection attack was in the BSD-Airtools package [4].

It is even possible to speed up the time it takes to capture the first ARP request. A *de-authenticate* message can be sent to a client in the network, telling him, he has lost contact with the base station. In some configurations we saw clients rejoining the network automatically at the same time flushing their ARP cache. The next IP packet sent by this client will cause an ARP request to look up the Ethernet address of the destination.

## 6 Our attack on WEP

The basic attack is straightforward. We use the methods described in Section 5 to generate a sufficient amount of key stream under different IVs. Initially we assume that a 104 bit WEP key was used. For every $\sigma_i$ from $\sigma_0$ to $\sigma_{12}$, and every recovered key stream, we calculate $A_i$ as described in equation 6 and call the result a *vote* for $\sigma_i$ having the value $A_i$. We keep track track of those votes in separate tables for each $\sigma_i$.

Having processed all available key streams, we assume that the correct value for every $\sigma_i$ is the one with the most votes received. The correct key is simply $\mathsf{Rk}[0] = \sigma_0$ for the first key byte and $\mathsf{Rk}[i] = \sigma_i - \sigma_{i-1}$ for all other key bytes. If the correct key was a 40 bit key instead of a 104 bit key, the correct key is just calculated from $\sigma_0$ to $\sigma_4$.

### 6.1 Key ranking

If only a low number of samples is available, the correct value for $\sigma_i$ is not always the most voted one in the table but tends to be one of the most voted. Figure 3 contains an example in which the correct value has the second most votes after 35.000 sessions. Instead of collecting more samples, we use another method for finding the correct key. Checking if a key is the correct key is simple, because we have collected a lot of key streams with their corresponding IV. We can just generate a key stream using an $\mathsf{IV}$ and a guessed key, and compare it with the collected one. If the method used for key stream recovery did not always guess the key stream right, the correct value just needs to match a certain fraction of some key streams.

For every key byte $\mathsf{K}[i]$, we define a set $M_i$ of possible values $\sigma_i$ might have. At the beginning, $M_i$ is only initialized with the top voted value for $\sigma_i$ from the table. Until the correct key is found, we look for an entry $\tilde{\sigma_i} \notin M_i$ in all tables having a minimum distance to the top voted entry in table $i$. We then add $\tilde{\sigma_i}$ to $M_i$ and test all keys which can now be constructed from the sets $M$ that have not been tested previously.

### 6.2 Handling strong keys

For equation 6 we assumed $\mathsf{S}_3$ to be a approximation of $\mathsf{S}_{3+i}$. This assumption is wrong for a fraction of the keyspace. We call these keys *strong keys*. For these keys, the value for $j_{i+3}$ is most times taken by $j$ in a iteration before $i+3$ and after 3. This results in $\mathsf{S}[j_{i+3}]$ being swapped with an unknown value, depending on the previous key bytes and the $\mathsf{IV}$. In iteration $i+3$, this value instead of $\mathsf{S}_3[j_{i+3}]$ is now swapped with $\mathsf{S}[i]$.

More formally, let $\mathsf{Rk}$ be a key and $\mathsf{Rk}[i]$ a key byte of $\mathsf{Rk}$. $\mathsf{Rk}[i]$ is a *strong key byte*, if there is an integer $l \in \{1, \ldots, i\}$ where

$$\sum_{k=l}^{i} (\mathsf{Rk}[k] + 3 + k) \equiv_n 0 \tag{10}$$

A key $\mathsf{Rk}$ is a *strong key*, if at least one of its key bytes is a *strong key byte*. On the contrary, key bytes that are not *strong key bytes* are called *normal key bytes* and keys in which not a single *strong key byte* occurs are called *normal keys*.

Assuming that $\mathsf{S}$ is still the identity permutation, the value 0 will be added to $j_{l+3}$ from iteration $l+3$ to $i+3$, making $j_{i+3}$ taking his previous value $j_{l+3}$. This results in the value $q_i$ from equation 8 being close to 0 and $\text{Prob}(\sigma_i = A_i)$ is very close to $\frac{1}{n}$.

Figure 3 shows the distribution of votes for a strong and a non strong key byte after 35.000 and 300.000 samples. It is easy to see that the correct value for strong key byte has not received the most votes of all key bytes any longer. An alternative way must be used to determine the correct value for this key byte. Our approach can be divided into two steps:

1. Find out which key bytes are strong key bytes
   If a key byte $\mathsf{Rk}[i]$ is a normal key byte, the correct value for $\sigma_i$ should appear with probability $\approx p_{\text{correct}_\text{i}}$ (see equation 9). We assume that all other values are equidistributed with probability $p_{\text{wrong}_\text{i}} = \frac{(1-p_{\text{correct}_\text{i}})}{n-1}$. If $\mathsf{Rk}[i]$ is a strong key byte we assume that all values are equidistributed with probability $p_{\text{equal}} = \frac{1}{n}$.
   Let $N_{i_b}$ the fraction of votes for which $\sigma_i = b$ holds. We calculate

$$\text{err}_{\text{strong}_\text{i}} = \sum_{j=0}^{n} \left(N_{i_j} - p_{\text{equal}}\right)^2 \tag{11}$$

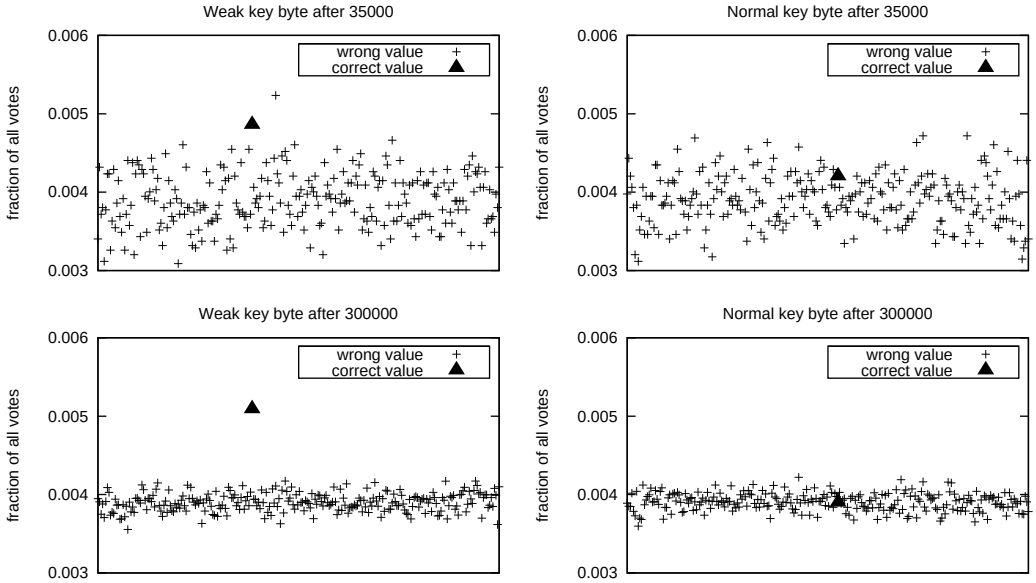
$$\text{err}_{\text{normal}_\text{i}} = \left(\max_b(N_{i_b}) - p_{\text{correct}_\text{i}}\right)^2 + \sum_{j=0, j\neq \text{argmax}_\text{b}(\text{N}_{\text{i}_\text{b}})}^{n} \left(N_{i_j} - p_{\text{wrong}_\text{i}}\right)^2 \tag{12}$$

   If enough samples are available, this can be used as a test, if $\text{err}_{\text{strong}_\text{i}}$ is smaller than $\text{err}_{\text{normal}_\text{i}}$, it is highly likely that key byte $\mathsf{Rk}[i]$ is a strong key byte. If only a small number of samples are available, $\text{err}_{\text{strong}_\text{i}} - \text{err}_{\text{normal}_\text{i}}$ can be used as an indicator, how likely it is, that key byte $\mathsf{Rk}[i]$ is a strong key byte.
2. Find the correct values for these key bytes
   Assuming that $\mathsf{Rk}[i]$ is a strong key byte and all values for $\mathsf{Rk}[0] \ldots \mathsf{Rk}[i-1]$ are known, we can use equation 10 and get the following equation for $\mathsf{Rk}[i]$:

$$\mathsf{Rk}[i] \equiv_n -3 - i - \sum_{j=l}^{i-1} (\mathsf{Rk}[j] + 3 + j) \tag{13}$$

   Because there at most $i$ possible values for $l$, we can try every possible value for $l$ and restrict $\mathsf{Rk}[i]$ to at most $i$ possible values (12 if $\mathsf{Rk}[i]$ is the last key byte for a 104 bit key). This method can be combined with the simple error correction method as described in Section 6.1. Instead of taking possible values for $\sigma_i$ from the top voted value in the table for key byte $i$, we ignore the table and use the values calculated with equation 13 for $\mathsf{Rk}[i]$ and assume that $\sigma_{i-1} + \mathsf{Rk}[i]$ was top voted in the table. Possible values for $\sigma_i$ for all assumed to be normal key bytes are still taken from the top voted values in their tables.

**Fig. 3.** Distribution of votes for a strong and a normal key byte




## 7 Experimental Results

We wrote an implementation using the parallelized computation as described in Section 4 and the error correction methods described in Section 6.1 and 6.2. At the beginning an upper bound on the number of keys to be tested is fixed (key limit). A limit of 1,000,000 keys seems a reasonable choice for a modern PC or laptop. Three different scenarios of attacks cover the most likely keys:

**scenario 1** tests 70% of the limit and uses the simple error correction as described in Section 6.1. As long as the set of possible keys does not exceed 70% of the key limit, a new value $v \notin M_i$ is added to a set $M_i$. The value $v$ is chosen to have minimal distance to the top voted entry.

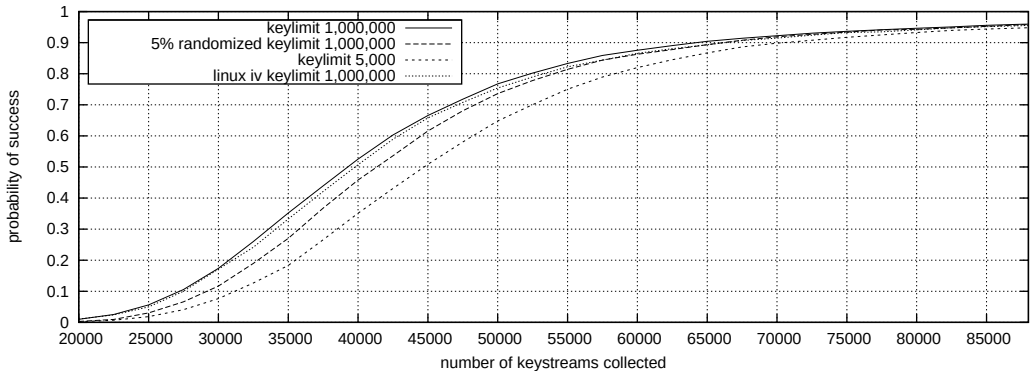
**scenario 2** tests 20% of the limit and uses the simple error correction method in combination with strong byte detection. We use the difference $\mathrm{err}_{\mathrm{strong_i}} - \mathrm{err}_{\mathrm{normal_i}}$ to determine which key byte is the most likely to be strong. We then use equation 13 to get possible values for this key byte. As long as the number of possible keys does not exceed 20% of the key limit all other key bytes are determined as in class 1.

**scenario 3** tests 10% of the limit and works like class 2, with the exception that 2 key bytes are assumed to be strong.

To verify a possible key $\mathsf{Rk}$ for correctness, 10 sessions ($\mathsf{IV}_i$, $\mathsf{X}_i$) are chosen at random. If the key stream generated using $\mathsf{IV}_i||\mathsf{Rk}$ is identical to $\mathsf{X}_i$ in the first 6 bytes for all sessions, we assume the key to be correct. If all three scenarios where unsuccessful, the attack is retried using just 10% of the key limit, this time under the assumption that the root key is a 40 bit key instead of a 104 bit one.

Figure 4 shows the result from a simulation, showing that a 50% success rate is possible using just 40,000 packets.

**Fig. 4.** Success rate




To test whether this attack works in a real world scenario we used the *aircrack-ng* tool suite. *aircrack-ng* contains *aireplay-ng*, a tool which is able to capture and replay 802.11 frames; for example, frames containing ARP requests or other kinds of packets that cause traffic on the target network. Additionally, *airodump-ng* can be used to capture and save the generated traffic including all needed IEEE 802.11 headers.

On a mixed IEEE 802.11 b/g network, consisting of cards with chipsets from Atheros, Broadcom, Intersil and Intel, we achieved a rate of 764 different captured packets per second, using *aireplay-ng* and a network card with a *PrismGT* chipset for injecting and an *Atheros* based card for capturing. This number might vary, depending on the chipsets and the quality of the signal. This allowed us to recover 40,492 key streams in 53 seconds. Additional 2 seconds can be added for deauthenticating one or all clients forcing them to send out new ARP requests. The final cryptographic computation requires 1 to 3 seconds of CPU-time, depending on the CPU being used. For a 104 bit key we were able to consistently and successfully recover keys from a packet capture in less than 3 seconds on a Thinkpad T41p (1.7 GHz Pentium-M CPU) – this includes reading and parsing the dump file, recovering key streams and performing the actual attack. On more recent multi-core CPUs we expect this figure can be brought down to less than a second with a parallelized key space search. This results in 54 to 58 seconds to crack a 104 bit WEP key, with probability 50%.

Main memory requirements for an efficient implementation are quite low. Our implementation needs less than 3 MB of main memory for data structures. Most of the memory is consumed by a bit field for finding duplicate IVs, this is 2MB in size. CPU-time scales almost linearly with the number of keys tested. By reducing the number of keys tested to 5,000, this attack is suiteable for PDA or embedded system usage too, by only reducing its success probability a little bit. The successrate with this reduced key limit is included in figure 4 with label *keylimit 5,000*.

### 7.1 Robustness of the attack

The key stream recovery method we used might not be always correct. For example, any kind of short packet (TCP, UDP, ICMP) might be identified as an ARP reply resulting in an incorrect key stream. To find out how our attack works with some incorrect values in key streams, we ran a simulation with 5% of all key streams replaced with random values. The

result is included in Figure 4, labeled "5% randomized key limit 1,000,000". Depending on the number of packets collected, the success rate is slightly reduced by less than 10%. If enough packets are available, there is no visible difference.

In all previous simulations, we assumed that IVs are generated independently, using any kind of pseudo random function. Some drivers in fact do use an PRNG to generate the IV value for each packet, however others use a counter with some modifications. For example the 802.11 stack in the Linux 2.6.20 kernel uses an counter, which additionally skips IVs which where used for an earlier attack on RC4 by Fuller, Mantin and Shamir which became known as FMS-weak-IVs. Using this modified IV generation scheme, the success rate of our attack (label *linux iv keylimit 1,000,000*) was slightly reduced by less than 5%, depending on the number of packets available. As before, there are no noticeable differences, if a high number of packets are available.

## 8 Further work

### 8.1 Better strong byte handling

Our current method for handling strong key bytes is not optimal. We are currently investigating some better methods for strong byte detection and strong byte correction. Because strong key bytes are relatively seldom in a randomly chosen key, this only had a minor priority for us. One possibility would be to additionally implement the unparalleled version of the Klein attack, which uses more CPU time when doing error correction, but is able to handle strong key bytes without problems.

### 8.2 Passive attack

Our current attack is a fast, but active one, and could thereby be detected by an Intrusion Detection System (IDS). A passive version would be interesting making the attack undetectable by any kind of network monitoring system. Such a passive version would require a more advanced key stream recovery method, because the first 16 bytes of an captured packet are not a fixed value, but contain some regular patterns. The cryptanalysis in a passive attack scenario is helped by the fact that Klein's method is robust to a small percentage of incorrectly guessed key streams, as we have shown in Section 7.1.

## 9 Conclusion

We have extended Klein's attack on RC4 and have applied it to the WEP protocol. Our extension consists in showing how to determine key bytes independently of each other and allows us to dramatically decrease the time complexity of a brute force attack on the remaining key bytes. We have carefully analyzed cases in which a straightforward extension of Klein's attack will fail and have shown how to deal with these situations.

The number of packets needed for our attack is so low that opportunistic attacks on this security protocol will be most probable. Although it has been known to be insecure and has been broken by a key-recovery attack for almost 6 years, WEP is still seeing widespread use at the time of writing this paper. While arguably still providing a weak deterrent against casual attackers in the past, the attack presented in this paper greatly improves the ease with which the security measure can be broken and will likely define a watershed moment in the arena of practical attacks on wireless networks.

## References


1. 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.
2. Stefan Dörhöfer. Empirische Untersuchungen zur WLAN-Sicherheit mittels Wardriving. Diplomarbeit, RWTH Aachen, September 2006. (in German).
3. 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.
4. David Hulton (h1kari). bsd-airtools. `http://www.dachb0den.com/projects/bsd-airtools.html`.
5. Andreas Klein. Attacks on the RC4 stream cipher. *submitted to Designs, Codes and Cryptography*, 2007.
6. 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.
7. 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.
8. Wi-Fi Alliance. Wi-Fi Protected Acccess (WPA). `http://www.wi-fi.org`.