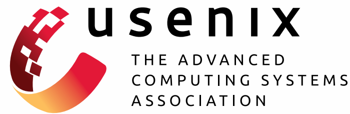


Predicting, Decrypting, and Abusing WPA2/802.11 Group Keys

Mathy Vanhoef and Frank Piessens, ***Katholieke Universiteit Leuven***

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/vanhoef

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

Predicting, Decrypting, and Abusing WPA2/802.11 Group Keys

Mathy Vanhoef
iMinds-DistriNet, KU Leuven
Mathy.Vanhoef@cs.kuleuven.be

Frank Piessens
iMinds-DistriNet, KU Leuven
Frank.Piessens@cs.kuleuven.be

Abstract

We analyze the generation and management of 802.11 group keys. These keys protect broadcast and multicast Wi-Fi traffic. We discovered several issues and illustrate their importance by decrypting all group (and unicast) traffic of a typical Wi-Fi network.

First we argue that the 802.11 random number generator is flawed by design, and provides an insufficient amount of entropy. This is confirmed by predicting randomly generated group keys on several platforms. We then examine whether group keys are securely transmitted to clients. Here we discover a downgrade attack that forces usage of RC4 to encrypt the group key when transmitted in the 4-way handshake. The per-message RC4 key is the concatenation of a public 16-byte initialization vector with a secret 16-byte key, and the first 256 keystream bytes are dropped. We study this peculiar usage of RC4, and find that capturing 2^{31} handshakes can be sufficient to recover (i.e., decrypt) a 128-bit group key. We also examine whether group traffic is properly isolated from unicast traffic. We find that this is not the case, and show that the group key can be used to inject and decrypt unicast traffic. Finally, we propose and study a new random number generator tailored for 802.11 platforms.

1 Introduction

In the last decennia, Wi-Fi became a de facto standard for medium-range wireless communications. Not only is it widely supported, several new enhancements also make it increasingly more performant. One downside is that (encrypted) traffic can easily be intercepted. As a result, securing Wi-Fi traffic has received considerable attention from the research community. For example, they showed that WEP is utterly broken [11, 42, 4], demonstrated attacks against WPA-TKIP [43, 45, 47, 41], performed security analysis of AES-CCMP [24, 39, 13], studied the security of the 4-way handshake [17, 18, 34], and so on. However, most research only focuses on the security of pairwise keys and unicast traffic. Group keys and group traffic have been given less attention, if mentioned at all.

In this paper we show that generating and managing group keys is a critical, but underappreciated part, of a modern Wi-Fi network. In particular we investigate the generation of group keys, their transmission to clients, and the isolation between group and unicast traffic. We discovered issues during all these phases of a group key's lifetime. To address some of our findings, we propose and implement a novel random number generator that extracts randomness from the physical Wi-Fi channel.

First we study the random number generator proposed by the 802.11 standard. Among other things, the Access Point (AP) uses it to generate group keys. Surprisingly, we find that it is flawed by design. We argue that implementing the algorithm as specified, results in an unacceptably slow algorithm. This argument is supported empirically: all implementations we examined, modified the generator to increase its speed. We demonstrate that these modified implementations can be broken by predicting the generated group key within mere minutes.

The generated group keys are transferred to clients during the 4-way WPA2 handshake. We found that it is possible to perform a (type of) downgrade attack against the 4-way handshake, causing RC4 to be used to encrypt the transmission of the group key. We analyze the construction of the per-message RC4 key and its effect on biases in the keystream. This reveals that an attacker can abuse biases to recover an 128-bit group key by capturing 2^{30} to 2^{32} encryptions of the group key, where the precise number depends on the configuration of the network.

Group keys should only be used to protect broadcast or multicast frames. In other words, pairwise and group keys should be properly isolated, and unicast packets should never be encrypted with a group key. An AP can enforce this by only sending, but never receiving, group addressed frames. However, all APs we tested did not provide this isolation. We demonstrate that this allows

FC	addr1	addr2	addr3	KeyID / PN	Data

Figure 1: Simplified 802.11 frame with a WPA2 header.

an attacker to use the group key to inject, and in turn decrypt, any traffic sent in a Wi-Fi network.

Finally, we propose and study a novel random number generation tailored for 802.11 platforms. It extracts randomness from the wireless channel by collecting fine-grained Received Signal Strength Indicator (RSSI) measurements. These measurements can be made using commodity devices even if there is no background traffic. We show our algorithm can generate more than 3000 bits per second, and even when an adversary can predict individual RSSI measurements with high probability, the output of the generator still remains close to uniformly random.

To summarize, our main contributions are:

- We show that the 802.11 random number generator is flawed, and break several implementations by predicting its output, and hence also the group key.
- We present a downgrade-style attack against the 4-way handshake, allowing one to recover the group key by exploiting weaknesses in the RC4 cipher.
- We show that the group key can be used to inject and decrypt any (internet) traffic in a Wi-Fi network.
- We propose and study a random number generator that extracts randomness from the wireless channel.

The rest of this paper is organized as follows. Section 2 introduces relevant parts of the 802.11 standard. We break the random number generator of 802.11 in Section 3. Section 4 presents a downgrade attack against the 4-way handshake, and attacks on its usage of RC4. In Section 5 we use the group key to inject and decrypt any frames, including unicast ones. In Section 6 we propose a new random number generator. Finally, we explore related work in Section 7, and conclude in Section 8.

2 Background

This section provides a background on the 802.11 protocol, the 4-way handshake, and the RC4 stream cipher.

2.1 The 802.11 Protocol

When a station wishes to transmit data, it needs to add a valid 802.11 header (see Figure 1). This headers contains the necessary MAC addresses to route the frame:

$$\text{addr1} = \text{Receiver MAC address}$$
$$\text{addr2} = \text{Sender MAC address}$$
$$\text{addr3} = \text{Destination MAC address}$$

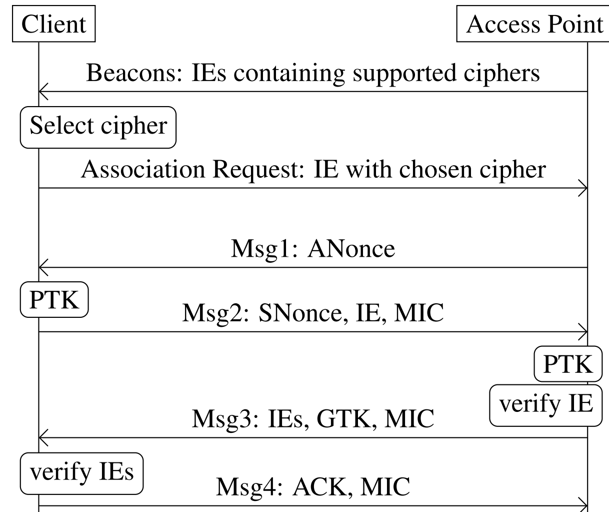


Figure 2: Discovering APs by listening to beacons, followed by the association and 4-way WPA2 handshake.

The Access Point (AP) forwards received frames to their destination, which is either a node on the wired network, or a Wi-Fi client. In frames received by a client, `addr1` should equal `addr3`, hence no further routing is required. For example, when a client sends an outbound IP packet, `addr1` equals the address of the AP, `addr2` contains his own address, and `addr3` equals the address of the router.

If a client wishes to transmit a broadcast or multicast frame, i.e., a group addressed frame, he first sends it as a unicast frame to the AP. This means `addr1` equals the address of the AP, and `addr3` equals the broadcast or multicast destination address. The AP then encrypts the frame using the group key if needed, and broadcasts it to all associated clients. This assures all clients within the range of the AP will receive the frame, even if certain stations are not within range of each other.

The Frame Control (FC) field contains, among other things, the `ToDS` and `FromDS` flags. The `ToDS` flag is set if the frame is sent from a client to an AP, and the `FromDS` flag is set if the frame is sent in the reverse direction. The fifth field in Figure 1 is only included when encryption is used, and contains the Key ID and Packet Number (PN). The PN prevents replay attacks. The 2-bit Key ID field is only used in group addressed frames, where it identifies which group key is used to protect and encrypt the frame.

2.2 Discovering APs and Negotiating Keys

Clients can discover APs by listening for beacons, which are periodically broadcasted by the AP (see Figure 2). These beacons contain the supported cipher suites of the AP in Information Elements (IEs). When a client wants to connect to an AP, and has selected a cipher to use, it starts by sending an association request to the AP. This

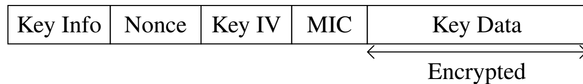


Figure 3: Simplified layout of EAPOL-Key frames.

request includes the selected cipher in an information element (IE). To prevent downgrade attacks, the client and AP will verify the received and selected IEs in the 4-way handshake. In this handshake, the client and AP also authenticate each other, and negotiate a Pairwise Temporal Key (PTK). The PTK is essentially the set of negotiated session keys. The first part of the PTK is called the Key Confirmation Key (KCK), and is used to authenticate handshake messages. The second part is called the Key Encryption Key (KEK), and is used to encrypt any sensitive data in the handshake messages. Finally, the third part is called the Temporal Key (TK), and is used to protect data frames that are transmitted after the handshake. To assure a new PTK is generated, both the client and AP first generate unpredictable, random nonces called SNonce and ANonce, respectively. The PTK is then derived from a shared secret or passphrase, the ANonce and SNonce, and the MAC addresses of the client and AP.

In the first message of the 4-way handshake, the AP sends the ANonce to the client (see Figure 2). On receipt of this message, the client calculates the PTK. In the second message, the client sends the SNonce, the IE representing the previously selected cipher, and includes a Message Integrity Code (MIC) calculated over the complete message. Note that the MIC is calculated using the KCK key contained in the PTK. After receiving Msg2 and the SNonce, the AP also derives the PTK. At this point both parties know the PTK, and all messages are authenticated using a MIC. Using the KCK and received MIC, the AP verifies the integrity of Msg2. The AP then checks whether the included IE matches the IE that was received in the initial association request. If these IEs differ, the handshake is aborted. Otherwise the AP replies with the Group Temporal Key (GTK), and its supported ciphers as a list of IEs. The client verifies the integrity of Msg3, and compares the included IEs with the ones previously received in the beacons. If the IEs differ, the handshake is aborted. Otherwise the client finishes the handshake by sending Msg4 to the AP.

Messages in the 4-way handshake are defined using EAPOL-Key frames, whose most important fields are shown in Figure 3. The Key Info field contains flags identifying which message this frame represents in the handshake. It also states which algorithm is used to calculate the MIC, and which cipher is used to encrypt the Key Data field (see Section 4). Note that the KCK key is used to calculate the MIC, and that the KEK key is used to encrypt the Key Data field. Finally, the Key IV field may contain an initialization vector (IV) to assure the Key Data field is always encrypted using a unique key. The most common usage of the Key Data field is to transport the group key (GTK), and to transfer any IEs.

Key Scheduling (KSA)

```
L = len(key)
j, S = 0, range(256)
for i in range(256):
  j += S[i] + key[i % L]
  swap(S[i], S[j])
return S
```

Keystream Output (PRGA)

```
S, i, j = KSA(key), 0, 0
while True:
  i += 1
  j += S[i]
  swap(S[i], S[j])
  yield S[S[i] + S[j]]
```

Figure 4: Implementation of RC4 in Python-like pseudo-code. All additions are carried out modulo 256.

2.3 The RC4 Stream Cipher

RC4 is a fast and well-known stream cipher consisting of two algorithms: a Key Scheduling Algorithm (KSA) and a Pseudo-Random Generation Algorithm (PRGA). Both are shown in Figure 4. The KSA takes as input a variable-length key, and generates a permutation S of the set $\{0,\dots,255\}$. This gradually changing permutation, combined with a public counter i and a private index j, form the internal state of the PRGA. In each algorithm, a swap operation is performed near the end of every round. We use the notations i_t, j_t, and S_t, for the indices i and j and the permutation S after round t. Rounds are indexed based on the value of i after the swap operation. Hence the KSA has rounds $t = 0,\dots,255$ and the PRGA has rounds $t = 1,2,\dots$. We let Z_r denote the keystream byte outputted at round r. Whenever it might not be clear whether we are referring to the KSA or PRGA, we use the notations S_t^{KSA} and S_t^{PRGA}, respectively.

Multiple biases have been found in the first few keystream bytes of RC4. These are called short-term biases. Arguably the most well known was found by Mantin and Shamir [30]. They showed that the value zero occurs twice as often at position 2 compared to uniform. In contrast, there are also biases that keep occurring throughout the whole keystream. We call these long-term biases. For example, Fluhrer and McGrew (FM) found that the probability of certain consecutive bytes deviate from uniform throughout the whole keystream [12]. Similarly, Mantin discovered a long-term bias towards the pattern $AB\mathcal{S}AB$, where A and B represent byte values, and $\mathcal{S}$ a short sequence of bytes called the gap [29]. Letting g denote the length of the gap, the bias can be written as follows:

$$\Pr[Z_r, Z_{r+1} = Z_{r+g+2}, Z_{r+g+3}] = 2^{-16}\left(1 + \frac{e^{(-4-8g)/256}}{256}\right)$$

Hence the longer the gap, the weaker the bias.

Listing 1: Random number generator as proposed by the 802.11 standard in Python-like pseudocode [21, §M.5].

```
1 def PRF-256(key, label, data):
2   R  = HMAC-SHA1(key, label + "\x00" + data + "\x00")
3   R += HMAC-SHA1(key, label + "\x00" + data + "\x01")
4   return R[:32]
5
6 def Hash(data):
7   return PRF-256(0, "Init Counter", data)
8
9 def NetworkJitter():
10   if ethernet traffic available:
11     return LSB(receive time of ethernet packet)
12   else:
13     Start 4-way handshake, stop after receiving Msg2
14     return LSB(Msg1.sent_time) + LSB(Msg2.rssi)
15       + LSB(Msg2.receive_time) + Msg2.snonce
16
17 def GenRandom():
18   local = "\x00" * 32
19   # Wait for Ethernet traffic or association, and
20   # loop until result is "random enough" or 32 times
21   for i in range(32):
22     buf = Hash(macaddr + currtime + local + i)
23     for j in range(32):
24       local += NetworkJitter()
25   return Hash(macaddr + currtime + local + "\x20")
```

3 Breaking the 802.11 RNG

In this section we argue that the Random Number Generator (RNG) of 802.11 is flawed, and break several implementations by predicting the generated group key.

3.1 The Proposed RNG in 802.11

The security enhancements amendment to 802.11, called 802.11i, includes a software-based RNG [22, §H.5.2]. It extracts randomness from clock jitter and frame arrival times. While the standard states the proposed algorithm is only expository, and real implementations should extend it with other sources of entropy, we found that several platforms directly implement it and even simplify it.

Listing 1 contains the proposed RNG as the function `GenRandom`. The outer for-loop first calculates a hash over the MAC address of the station, the current time, the `local` variable, and the loop counter. Then it makes multiple calls to `NetworkJitter` in order to collect randomness from the arrival times of Ethernet or Wi-Fi frames. Here `LSB` returns the least significant byte of a timestamp. Note the comment on line 20, which is copied almost verbatim from the standard. It instructs to either run the outer loop 32 times, or until the `lcoal` variable is “random enough”. No clarification is made on what this exactly means. The standard also mentions that the variable `currtime` can be set to zero if the current time is not available. However, there is no discussion on how this impacts the RNG, e.g., whether additional iterations of the outer for-loop should be executed. Additionally, the standard does not mandate a minimum resolution for the timestamps that are used. It only states that the send and receive timestamps of frames should use the highest resolution possible, *preferably* 1 ms or better. Finally, the RNG is executed on demand, i.e., there is no state saved between two invocations of `GenRandom`.

3.2 Analysis

A careful inspection of the RNG shows it is ill-defined and likely insecure. One problem lies with the `NetworkJitter` function, which is called 256 times by `GenRandom`. First, the if test on line 10 is ambiguous. If it checks whether there was Ethernet traffic in the last x seconds, repeated calls will probably operate on the same Ethernet packet, and the function will return the same data. On the other hand, if this test implies monitoring the Ethernet interface for x seconds, this might cause a total delay of $256 \cdot x$ seconds. Furthermore, the value of x is not given. In any case, either calling `NetworkJitter` implies waiting a significant amount of time until there is new traffic, or repeated calls return the same value.

When the second clause of the if statement on line 10 is taken, the arrival times of frames transmitted during the 4-way handshake are used. Specifically, it mentions to initiate the 4-way handshake. This is something only an AP can do when a client is trying to connect to this AP (see Figure 2). Therefore the proposed algorithm is only usable by APs. We also remark that if the AP were to constantly abort the handshake after receiving message 2 (see line 13), most clients will blacklist the AP for a certain period. During this period, the client will no longer attempt to connect to the network. Hence it becomes infeasible to initiate and abort 256 4-way handshakes, which is something the random number generator is supposed to do. We conclude that the function `NetworkJitter` is unusable in practice. Based on this we conjecture, and empirically confirm in Section 3.4, that vendors will not implement this function.

Another design flaw is that no state is kept between subsequent calls to `GenRandom`. Hence its output depends only on a small amount of network traffic and timestamp samples. A better design is to collect randomness in a pool, and regularly reseed this pool with new randomness. When done properly, this protects against permanent compromise of the RNG, iterative guessing attacks, backtracking attacks, and so on [25, 3, 9].

We conclude that the proposed RNG is questionable at best. Either it returns bytes having a low amount of entropy, or calling it will incur significant slowdowns. In Section 3.4 we will show that in practice this construction results in defective and predictable RNGs.

3.3 Generation of the Group Key

The 802.11 standard defines, but does not mandate, a key hierarchy for the generation of group keys [21, §11.6.1]. This hierarchy is described in Listing 2, where `macaddr` denotes the MAC address of the AP, and `currtime` is either the current time or zero. The `on_startup` function is executed at boot time, and generates a random auxiliary key called the Group Master Key (GMK). Additionally, it initializes the `key_counter` variable to a pseudo-random value [21, §11.6.5]. Actual group keys, called Group Temporal Keys (GTKs), are derived from the GMK and `key_counter` using a Pseudo-Random Function (PRF) in `new_gtk`. The length of the generated GTK depends on the cipher being used to protect group traffic. If TKIP is used, the GTK is 32 bytes long. If CCMP is used, the GTK is 16 bytes long. This implies the PRF has to generate either 128 or 256 bits of keying material, depending on the configuration of the network. Hence we use the function name `PRF-X`, where the value of *X* depends on the amount of requested keying material. Note that the implementation of `PRF-256` is shown in Listing 1, and that `PRF-128` closely resembles this function. The latest standard also states [21, §11.6.1.4]:

> "The GMK is an auxiliary key that may be used to derive a GTK at a time interval configured into the AP to reduce the exposure of data if the GMK is compromised."

However, this makes no sense: there is no point in introducing a new key to reduce the impact if that key itself leaks. Curiously, we found that older versions of the standard did not contain this description of the GMK. Instead, older versions stated that the GMK may be reinitialized to reduce the exposure of data in case the current value of the GMK is ever leaked.

Most implementations renew the GTK every hour by calling a function similar to `new_gtk`. More importantly, the `key_counter` variable is also used to initialize the Key IV field of certain EAPOL-Key frames (see Figure 3). After using the value of `key_counter` for this purpose, it is incremented by one. Since these IVs are public values, the value of `key_counter` is known by adversaries. We found that some implementations even use `key_counter` to generate nonce values during the 4-way handshake, though this is not recommended as it may enable precomputation attacks [21, §8.5.3.7].

One major disadvantage of the proposed key hierarchy, is that fresh entropy is never introduced when generating a new group key in `new_gtk`. Hence, once the value of `GMK` has been leaked, or recovered by an attacker, all subsequent groups keys can be trivially predicted.

Since the standard assumes that `GenRandom` provides cryptographic-quality random numbers, there appears to be no advantage in using this key hierarchy. Instead, the AP can directly call `GenRandom` to generate new group keys. Some consider this key hierarchy a relic from older 802.11 standards, which did not yet require that devices must implement a strong RNG [20]. Perhaps the only (unintended) advantage this construction has, is that the *first* group key is now determined by two calls to `GenRandom`, instead of only one call. Hence, if an adversary is trying to attack a weak implementation of `GenRandom`, he has to predict its output twice (see Section 3.4). Nowadays implementations are allowed to directly generate a random value for the GTK [21, §11.6.1.4], though many platforms still implement the proposed group key hierarchy (see Section 3.4).

Listing 2: Python-like pseudocode describing the group key hierarchy (and generation) according to the 802.11 standard.

```
1 def on_startup():
2   GMK, key = GenRandom(), GenRandom()
3   buf = macaddr + currtime
4   key_counter = PRF-256(key, "Init Counter", buf)
5
6 def new_gtk():
7   gnonce = key_counter++
8   buf = macaddr + gnonce
9   GTK = PRF-X(GMK, "Group key expansion", buf)
```

3.4 Practical Consequences

We now study the RNG of real 802.11 platforms. First we focus on popular consumer devices. To estimate the popularity of a specific brand, we surveyed wireless networks in two Belgian municipalities. We were able to recognize specific brands based on vendor-specific information elements in beacons. We detected 6 803 networks, and found that MediaTek- and Broadcom-based APs alone covered at least 22% of all Wi-Fi networks. We will focus on both because of their popularity. Additionally we examine Hostapd for Linux. Finally, we study embedded systems by analysing the Open Firmware project. We found that, apart from Hostapd, all these platforms produce predictable random numbers.

3.4.1 MediaTek-based Routers

Access points with a MediaTek radio use out-of-tree Linux drivers to control the radio[1]. These drivers directly manage the 4-way handshake and key generation. They implement the 802.11 RNG as shown in Listing 1, but do not call `NetworkJitter`. This strengthens our hypothesis that this function is infeasible to implement in

[1] Available from www.mediatek.com/en/downloads1

practice. It also means the only source of randomness is the current time, for which it uses the *jiffies* counter of the Linux kernel. This counter is initialized to a fixed value at boot, and incremented at every timer interrupt. The number of timer interrupts per second is configured at compile time and commonly lies between 100 and 1 000. Hence it is a coarse grained timestamp, meaning the `currtime` variable likely has the same value each time it is sampled in `GenRandom`. That is, the current time is the only random source being used, and provides little entropy.

The group key hierarchy is implemented according to the 802.11 standard, with one exception. Instead of initializing `gnonce` to `key_counter` in line 7 of Listing 2, it generates a new value using `GenRandom`, and assigns the result to `gnonce`.

We show that this RNG is flawed by predicting the group key generated by an Asus RT-AC51U. A similar approach can be followed for other routes that also use a MediaTek radio. The first step is to predict the GMK generated at boot. By recompiling the firmware, and printing out the jiffies values that were used at the start and end of an invocation of `GenRandom`, we observe that it uses at most two different values. Note that we printed these values out only after calling `GenRandom`, to assure we did not noticeably influence the used jiffies values. Hence the jiffies values is incremented at most by one while executing `GenRandom`. Since this increment may happen in any of the 32 loops, `GenRandom` can result in total 32 possible values if the initial jiffies value is known. If AES-CCMP is used to protect group traffic, the initial jiffies value when generating the GMK lies in the range $[2^{32} - 72\,889, 2^{32} - 72\,884]$. If WPA-TKIP is used, it lies in $[2^{32} - 73\,067, 2^{32} - 73\,061]$. The number of attached USB or Ethernet devices, amount of Ethernet traffic, or other Wi-Fi options, did not impact these estimates. Since it is trivial to determine whether AES or TKIP is used, and less than 10 possible initial values are used in both cases, we end up with at most $32 \cdot 10$ possible values for the GMK.

The second step is to estimate the jiffies count when the GTK, i.e., group key, was generated. By default, a new group key is generated every hour. Hence, if we know the uptime of the router, we can determine when the current group key was generated. Conveniently, beacons leak the uptime of a device in their timestamp field. This field is used to synchronize timers between all stations [21, §10.1], and is generally initialized to zero at boot. Hence its value corresponds to the uptime of the router. From this we can estimate the jiffies counter's value at the time the group key was generated. However, as the device keeps running, clock skew will affect our prediction. By logging jiffies values, we observed that the clock skew over one month made our prediction off by at most 4 500 jiffies. Therefore, even after an uptime of year, our prediction of the jiffies value will only be off by roughly 50 000. In other words, we conjecture that the prediction after a year will be off by at most 200 seconds.

We created an OpenCL program to search for the group key on a GPU. It tests candidate keys by decrypting the first 8 bytes of a packet, and checking if they match the predictable LLC/SNAP header. If the targeted router has been running for one year, it has to test $320 \cdot 50000 \cdot 32 \approx 2^{29}$ candidates to recover the group key. However, testing each key is rather costly, as it involves calculating $33 \cdot 4$ SHA-1 hashes to derive the group key, and we must then decrypt the first 8 bytes of the packet to verify the key. Nevertheless, on our NVIDIA GeForce GTX 950M, it takes roughly two minutes to test all 2^{29} candidates and recover the GTK. We confirmed this by successfully predicting several group keys generated by our Asus RT-AC51U, when it had an uptime of more than a month. We conclude the group key generated by a MediaTek driver can be brute-forced using commodity hardware in negligible time.

3.4.2 Broadcom Network Authentication

The network access server of Broadcom implements the 4-way handshake, including the necessary key generation. It implements the group key hierarchy according to the 802.11 standard (see Listing 7). Additionally, it uses the `key_counter` variable to initialize the Key IV field of EAPOL-Key frames. However, it does not implement the RNG as proposed in the 802.11 standard. Instead, the RNG it uses depends on the kernel used by the device.

When running on a VxWorks or eCos kernel, random numbers are generated by taking the MD5 checksum of the current time in microsecond accuracy. Hence random numbers are straightforward to predict. And since the output of MD5 is used, only 16 bytes of supposedly random data is generated in every call. One widely used device that uses a VxWorks kernel, is version 5 or higher of the popular WRT54G router. Furthermore, the Apple AirPort Extreme also uses a VxWorks kernel. To predict the group key generated by these devices, we only have to predict the value of `GMK`. Recall that the value of `key_counter` is leaked in the Key IV field of certain EAPOL-Key fields, and can simply be passively sniffed. Since `GMK` is a 32-byte value, it is initialized by calling the random number generator twice. In order to predict the output of these two calls, we must first determine the time at which the group key was generated based on the uptime of the router. Similar to the MediaTek case, the uptime can be derived from the timestamp field in beacons. Assuming we can estimate time at which the group key was generated with an accuracy of one second, and that the timestamp in the next call to the RNG differs by

Listing 3: Generation of random nonces by the Open Firmware project in Python-like pseudocode.

```
1 def on_system_boot():
2   rn = lcg_next(milliseconds since boot)
3   data = macaddr + rn
4   rn = lcg_next(rn)
5   nonce = PRF-256(rn, "Init Counter", data)
6
7 def lcg_next(rn):
8   return rn * 0x107465 + 0x234567
9
10 def compute_next_snonce():
11   nonce += 1
12   return nonce
```

at most 10 ms, we have to test $1\,000\,000 \cdot 10\,000 \approx 2^{33}$ keys. We implemented an OpenCL program to simulate this search, and on our GeForce GTX 950M, it takes around 4 minutes to test all candidate keys. Hence the generated group keys by this RNG can be predicted using commodity hardware.

When running on a Linux kernel, random bytes are read from `/dev/urandom`. This is problematic since, on routers and embedded devices, `/dev/urandom` is commonly predictable at boot [19]. And since entropy for the group keys is only collected at boot (see Section 3.3), this again means all groups keys may be predictable.

3.4.3 The Linux Hostapd Daemon

Hostapd implements the 802.11 group key hierarchy as shown in Listing 2. However, when generating a new group key using a function similar to `new_gtk`, it also samples and incorporates new entropy. Additionally, Hostapd does not implement the 802.11 RNG. Instead, it generates keys by reading from `/dev/random`. In case insufficient entropy is available, it will re-sample from `/dev/random` when the first client is attempting to connect. In case there still is not enough entropy available, the client is not allowed to connect All combined, this means the keys used by Hostapd should be secure.

3.4.4 Open Firmware (OpenBoot)

The Open Firmware project, previously called OpenBoot, is a free and open source boot loader programmed in Forth[2]. Most notably it is used in the One Laptop Per Child project. Interestingly, it provides basic but secure Wi-Fi functionality during the early stages of the boot process. Since at this stage no operating system is loaded, we consider it an ideal candidate to investigate how vendors implement RNGs in a constrained (embedded) environment.

[2] Available from `svn://openbios.org/openfirmware`

Currently, the Wi-Fi module of Open Firmware only provides client functionality. Therefore we focus on the generation of random nonces during the 4-way handshake. Listing 3 illustrates how Open Firmware generates these nonces. Summarized, when loading the Wi-Fi module, a random initial nonce is generated, and this nonce is incremented whenever it is used. In this regard, the algorithm follows the 802.11 standard [21, §11.6.5]. However, the generation of the initial random nonce is very weak. It takes the uptime of the device in number of milliseconds, runs this twice through a linear congruential generator, combines it with its own MAC address, and finally expands this data using a Pseudo-Random Function (PRF). All this information can be predicted or brute-forced by an adversary.

We attribute this weak construction to a careless implementation, and treat it as an indication that a better design is to let the Wi-Fi chip generate random numbers itself. Users can then query the Wi-Fi chip when new randomness is needed. In Section 3 we demonstrate how a strong random number generator can be implemented using commodity Wi-Fi devices.

4 RC4 in the 4-Way Handshake

In this section we present a (type of) downgrade attack against the 4-way handshake. As a result, RC4 is used to encrypt sensitive information in the handshake. We present two attacks against the usage of RC4 in the handshake, and show how it allows an attacker to recover the group key. We also determine the performance of our attacks, and propose countermeasures.

4.1 Downgrading to RC4

When inspecting the 4-way handshake in Figure 2, we can see that the AP sends the group key (GTK) to the client before the client verifies the IEs of the AP. Recall that these IEs contain the supported cipher suites of the AP, which are advertised in plaintext beacons. In other words, the client can only detect that a downgrade attack has occurred *after* the AP has transmitted the group key in Msg3. This is problematic because, if multiple ciphers can be used to protect the handshake, an adversary can try to perform a downgrade attack to induce the AP into encrypting and transmitting the group key using a weak cipher.

Interestingly, the 4-way handshake can indeed be protected by several cipher suites [21, §11.6.2], meaning a downgrade attack is possible. More specifically, the cipher suite that is used to protect the handshake is determined by two settings that may, or may not, be requested by the client in its association request (see Figure 2). In

Table 1: Cipher suites used in the 4-way handshake.

Selected Options	Ciphers Used
Fast Transition (FT)	AES-CMAC, AES key wrap
CCMP without FT	HMAC-SHA1, AES key wrap
TKIP without FT	HMAC-MD5, RC4

particular, when support for fast network transitions is requested, AES-CMAC and and NIST AES key wrap are used to protect messages in the 4-way handshake. Otherwise, the cipher used to protect the handshake depends on the pairwise cipher that will be used to protect normal data frames transmitted after the handshake. In case CCMP will be used, the 4-way handshake uses HMAC-SHA1 and NIST EAS key wrap. More troublesome, if TKIP will be used, then HMAC-MD5 and RC4 are used to authenticate and encrypt data, respectively. An overview of this selection process is shown in Table 1.

Our idea is now to create a rogue AP that only advertises support for TKIP. Hence victims wanting to connect to the AP will use TKIP, and in turn the group key transmitted in the 4-way handshake will be encrypted using RC4. This works because the client will only detect the downgrade attack after receiving message 3. However, to make the AP send message 3, it must first receive and successfully verify the integrity of message 2. If its integrity cannot be verified, which is done using the negotiated session keys, the AP will not continue the handshake. Since the session keys depend on the MAC addresses of the client and AP, it means we must create a rogue AP with the same MAC address as the real one. Fortunately this is possible by performing a channel-based man-in-the-middle attack [46]. Essentially, the attacker clones the AP on a different channel, and forwards packets to, and from, the real AP. The MAC addresses in forwarded frames are not modified. This assures the station and AP will generate the same session keys, meaning the AP will successfully verify the authenticity of message 2. This man-in-the-middle position allows the attacker to reliably manipulate messages. In particular, it will use this position to modify the beacons and probe responses so it seems the AP only supports (WPA-)TKIP. Hence the client will be forced to select TKIP, causing the AP use to RC4 for encrypting the group key.

We tested this downgrade-style attack against a network that advertised both support for TKIP and CCMP. Since the rogue AP only advertised support for TKIP, the victim indeed selected TKIP. We then confirmed that the AP encrypts the group key using RC4, and that the client detected our attack only after receiving message 3.

Interestingly, the 4-way handshake uses RC4 in a rather peculiar manner [22, §8.5.2j]. The per-message RC4 key is the concatenation of the 16-byte Initialization Vector (IV) and the 16-byte Key Encryption Key (KEK). Additionally, the initial 256 keystream bytes are dropped. This construction is similar to the one used by WEP, except that the IV is longer, and that some initial keystream bytes are dropped. Interestingly, using a longer IV likely weakens this per-message key construction [28, 36], while dropping the initial keystream bytes should strengthen it [33, 28]. In the next two sections, we analyze the impact of this peculiar combination.

4.2 Recovering the Key Encryption Key

We first examine whether it is possible to perform a key recovery attack similar to those that broke WEP [11, 42]. In general, these attacks are applicable if a public IV is prepended (or appended) to a fixed secret key. This matches the construction of the per-message key K in the 4-way handshake, where the public 16-byte IV is prepended to the secret but static 16-byte KEK key. More formally, the per-message key is constructed as follows:

$$K = IV \mid\mid KEK$$

Although the first 256 keystream bytes of RC4 are dropped, Mantin showed this does not prevent key recovery attacks [28]. We will adapt Mantin's attack to the 4-way handshake, an study whether it is feasible to perform this attack in practice.

Similar to the original FMS attack [11], Mantin's extension of the FMS attack uses an iterative process to recover the key K [28]. That is, each iteration assumes the first x bytes of K are known, and attempts to recover the next key byte $K[x]$. In the first step of each iteration of Mantin's new attack, keystreams are collected that were generated with an IV for which the condition $S^{KSA}_{x-1}[1] = x$ holds. We call these *applicable* IVs. Mantin proved that applicable IVs leak information about the key byte $K[x]$ through the following relation [28]:

$$\Pr[\, K[x] = S^{-1}_{x-1}[i_{257} - z_{257}] - j_{x-1} - S_{x-1}[x]\,] \approx 1.1 \cdot 2^{-8} \quad (1)$$

This relation can be used to recover $K[x]$ with a simple voting mechanism as follows. Each applicable IV casts a vote for a certain value through equation (1). After all IVs are processed, the value with the most votes is assumed to be the value of $K[x]$. Unfortunately, this has the downside that a single incorrect guess for any byte means the complete key K is also wrong. To mitigate this, we pick the C most likely values for each byte, and construct C^{16} candidate values for K. Each candidate can be tested based on the captured IVs and corresponding keystreams. We simulated this attack against the 4-way handshake, with as goal to determine how many applicable IVs have to be capture to recover the *KEK* key. The result of this simulation is shown in Figure 5. To obtain

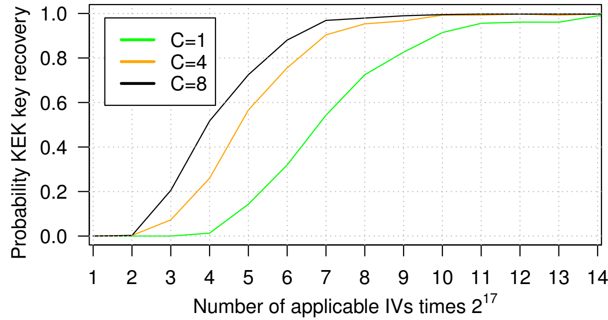


Figure 5: Probability of finding the 16-byte KEK key given the number of applicable IVs and the branch factor C.

a 80% success ratio of recovering the *KEK*, roughly 2^{21} applicable IVs have to be collected. Note that to execute the attack, we must be able to determine the value of z_{257}. Fortunately, this is possible by relying on the predictable IEs that are located at the start of the EAPOL Key Data field, meaning we can derive the keystream at these initial positions.

We now determine how much effort it takes to collect the required number of applicable IVs. First notice that the most likely way the condition $S^{KSA}_{x-1}[1] = x$ is satisfied, is when the value x is swapped into position 1 at round 1 of the KSA (recall Listing 4). Or more formally, that $j_1 = x$, since $K[x]$ is likely not modified in the first round. Indeed, the 15 other rounds only affect position 1 if j ever equals 1. Assuming a random initial IV is used, this will not happen with a probability of $\left(\frac{255}{256}\right)^{15} = 0.94$. At round one, $j_1 = K[0] + 1 + K[1]$ (we assume $K[0] \neq 1$ which holds with high probability). Hence, with high probability, an IV is applicable when $K[0] + K[1] = x - 1$.

The 802.11 standard states that a station must generate an initial random IV at startup, and increment this IV after it is used in a message [21, §11.6.5]. In other words, the IV is used as a counter. However, it does not specify whether little or big endian counters must be used. Our experiments indicate that most devices we use the IV as a big endian counter. Fortunately, from a defenders perspective, this means that generating the required number of applicable IVs takes an enormous amount of time. Assuming that the condition $K[0] + K[1] = x - 1$ does not hold, we must wait until $K[1]$ has been incremented sufficiently many times. However, only every $256^{14} = 2^{112}$ IVs does the value of $K[1]$ change. Clearly, this means that collecting the required number of IVs is infeasible when the AP uses a big endian counter. If the IV is generated by a little endian counter, the condition $K[0] + K[1] = x - 1$ is generally satisfied every 256th message. This means around $256 \cdot n$ messages must be collected in order to have roughly n applicable IVs for all iterations.

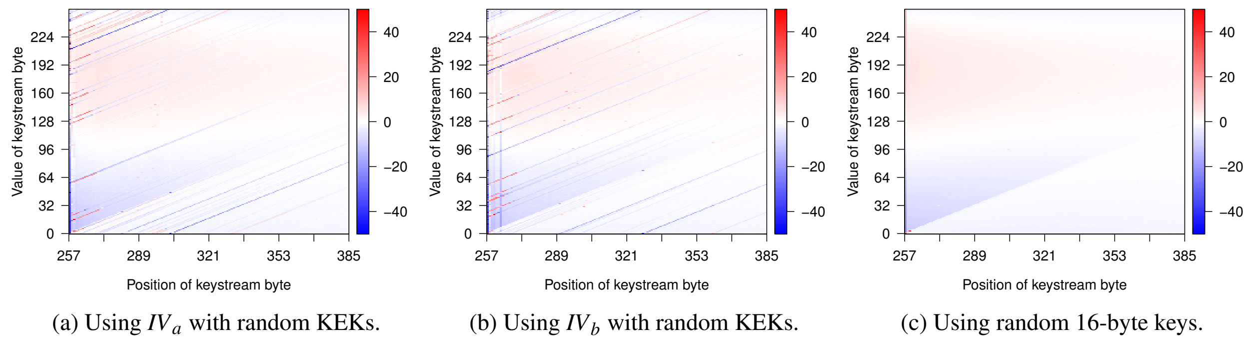
While it is possible to generate many handshakes by forcibly disconnecting clients, new handshakes will use a different KEK key. Since our attack assumes that the KEK is constant, this is not an option. The only method we identified to make the AP send several handshake messages protected by the same KEK, is by not acknowledges them, and letting the AP retransmit them. Note that each retransmission uses a new IV. Unfortunately, only a few messages will be retransmitted before the AP gives up and aborts the handshake process. In the next Section we present an attack that does tolerate frequent changes of the KEK key. We conclude that the 4-way handshake, as defined in the 802.11i standard, is vulnerable to key recovery attacks. However, these attacks seem difficult to pull off against popular implementations.

4.3 Plaintext Recovery Attacks

We now turn our attention to plaintext recovery attacks, where an adversary targets information that is repeatedly encrypted under different RC4 keys. Previous work on RC4 has shown that these types of attacks can be very successful, with attacks against TLS and TKIP being on the verge of practicality [2, 47]. In particular, the attack against WPA-TKIP by Paterson et al. [37] is fairly similar to our scenario. They showed that for WPA-TKIP, the public 3-byte prefix of the per-message RC4 key induces large, prefix-dependent, biases into the RC4 keystream [37, 15]. An adversary can precompute these prefix-dependent biases, and mount a powerful plaintext recovery attack against the first few bytes encrypted by RC4. This inspired us to investigate whether the public 16-byte IV used in the 4-way handshake also induces IV-dependent biases, even though the first 256 keystream bytes are dropped. Hence we examine the biases induced by the public IV contained in EAPOL frames, and then demonstrate through simulations that these can be used to recover the group key.

It is impossible to empirically investigate every possible IV, since this would mean inspecting 2^{128} values. Instead, we initially generated detailed keystream statistics for four randomly selected IVs. This indicated that large, IV-dependent biases indeed persist in the keystream, even after the first 256 bytes (which are dropped). Motivated by this result, we took the all-zero initialization vector IV_0, and investigated how changing the values at each specific position in the IV influences the keystream distribution. Changing a byte at position x to value y is denoted by $IV_0[x] = y$. The generation of all datasets took more than 13 CPU years.

Figure 6a and 6b show the keystream distribution for the initialization vectors IV_a and IV_b, respectively. The



(a) Using IV_a with random KEKs.

(b) Using IV_b with random KEKs.

(c) Using random 16-byte keys.

Figure 6: Biases in the RC4 keystream when concatenating a fixed 16-byte IV with a random 16-byte key (here called KEK key), and when using random 16-byte keys. Each points encodes a bias as the number $(pr - 2^{-8}) \cdot 2^{24}$, capped to values in $[-50, 50]$, with pr the empirical probability of the keystream byte value (y-axis) at a given location (x-axis).

distributions of IV_c and IV_d behave similarly. Their values were randomly generated in Python and are:

$IV_a =$ `0x2fe931f824ef842bf262dbca357bb31c`

$IV_b =$ `0x48d9859f9fa08bb1599744a20491dd49`

$IV_c =$ `0x6c1924761b03faf8decc0dfc09dd3078`

$IV_d =$ `0xe31257489cbe7d91e5365286c26f5023`

These keystream distributions were generated using 2^{45} RC4 keys for each IV. For comparison, Figure 6c and Figure 9c shows the keystream distribution for fully random 16-byte keys, generated using roughly 2^{47} RC4 keystreams [47]. We observe that the initialization vector induces strong biases that are visible as straight lines, even after position 256. By comparing these biases with the keystream distribution of 16-byte random RC4 keys in Figure 6c, we can conclude that the biases represented by the light and red background, are not caused by the specific IV values. Instead, they appear inherent to RC4.

We also generated 16 datasets, where in each dataset the value at one specific position in the all-zeros IV is changed. That is, for $0 \leq x \leq 15$, we generated datasets for the vectors $IV_0[x] = y$, where y ranges between 0 and 255. Each dataset was generated using 2^{43} keys, resulting in rather noisy distributions. Nevertheless, an inspection of these datasets confirmed that each IV induces specific biases, visible as straight lines in our graphs. A more detailed discussion of these biases is out of scope, and is left as future work.

We now use the IV-dependent biases to recover repeated plaintext, in order to get an indication of how well a plaintext recovery attack works against the 4-way handshake. This is done by combining the precomputed keystream distributions with captured ciphertexts, in order to calculate likelihood estimates for each plaintext value. The actual plaintext value is then assumed to be the one with the highest likelihood. Since our goal is mainly to evaluate the performance of the resulting attacks, we refer to previous work for the technicalities behind these calculations [2, 37, 47]. In particular, we implemented the binning algorithm proposed by Paterson et al. [37], and the single-byte candidate generation algorithm proposed by Vanhoef and Piessens [47]. To keep the computations feasible, we assumed that half of the captured ciphertext were generated by IV_a, and the other half by IV_b. For the binning algorithm of Paterson et al., Figure 7 shows the probability of correctly decrypting a byte. For the candidate generation algorithm, which returns a list of plaintext candidates for a sequence of bytes, we first need to determine at which position the group key is stored in the EAPOL Key Data field.

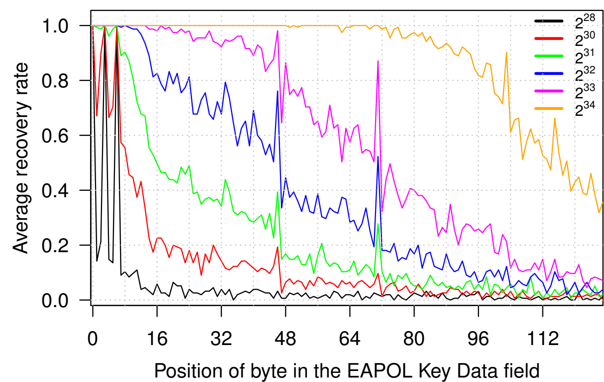


Figure 7: Success rate of decrypting a byte in the EAPOL Key Data field, in function of the byte position and number of collected ciphertexts. The legend shows the total number of ciphertexts used, where half of the ciphertexts were generated using IV_a, and the other half using IV_b.

The location of the group key depends on which cipher suites are supported. If only TKIP is supported in a RSN network, it starts at position 30. In contrast, if both TKIP and AES are supported, and if the older WPA informa-

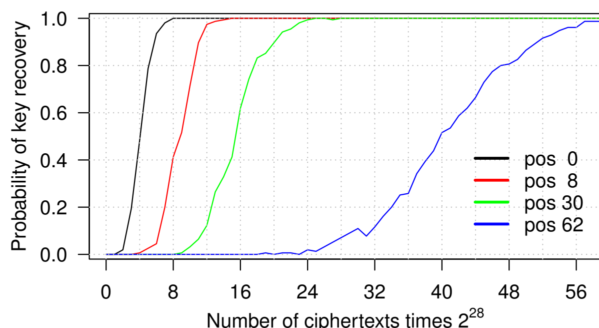


Figure 8: Probability of recovering a 16-byte key using short-term biases and 2^{26} candidates. The legend shows its starting position in the EAPOL key data field.

tion elements are included in addition to the RSN IEs[3], the group key starts at position 62. For other configurations, the group key is located somewhere between position 30 and 62. The probability of recovering a 16-byte key at these positions, in function of the number of captured ciphertexts (handshakes), is shown in Figure 8. We remark that if the group key starts at position 62, attacks that exploit Mantin's *ABSAB* bias become more efficient [29, 6]. This is because the 62 bytes that precede the group key are predictable, and hence an attack similar to the one that broke RC4 in TLS can be launched [47].

Finally, in principle it is also possible to attack group key update messages, since the 802.11 standard does not mandate that these messages should be encrypted using the pairwise cipher [21, §11.6.7]. Group key updates use EAPOL frames, and are sent when the AP generates a new group key. Interestingly, in these messages the group key is either located at position 8, or at position 0. The success rate of recovering a 16-byte key at these positions is shown in Figure 8. For example, if the key starts at position 8, roughly 2^{31} encryptions of the GTK have to be captured in order to decrypt it. Since in this case there is little surrounding known plaintext, it is the best attack an adversary can launch [2, 47, 6].

4.4 Countermeasures

To prevent the downgrade attack, APs should disable support of WPA-TKIP. Even when an adversary creates a rogue AP advertising TKIP, the real AP will reject any request for TKIP, and hence will never use RC4 in the 4-way handshake. Similarly, clients should not connect to a network using WPA-TKIP.

[3]Early implementations based on the draft 802.11i standard use WPA IEs, instead of RSN IEs as used in modern networks.

5 Abusing the Group Key

In this section we show that the group key can be used to decrypt and inject any traffic, including unicast traffic.

5.1 Injecting Unicast Frames

First we explain how to inject unicast traffic using the group key. This is not possible by simply encrypting a unicast frame with the group key, and setting the KeyID field to the id used by the group key. The receiver always uses pairwise keys to decrypt unicast frames, and ignores the KeyID value (recall Section 2.1). Furthermore, it is not possible to encapsulate unicast IP packets in group addressed frames. Indeed, RFC 1122 states that stations should discard unicast IP packets that were received on a broadcast or multicast link-layer address [5, §3.3.6]. However, this check is only performed when the packet is passed on to the IP layer. Since an AP does not operate at the IP layer, but on the MAC layer, it does not perform this check. Inspired by this observation, we encrypt the unicast IP packet using the group key, and send it to the AP. For the three address fields in the frame we use the following values:

addr1 = `FF:FF:FF:FF:FF:FF`
addr2 = Spoofed sender MAC address
addr3 = Spoofed destination MAC address

Although in a normal network the AP never processes group addressed frames, we found that APs can be forced to process our injected broadcast packet by setting the ToDS bit in the Frame Control (FC) field. To assure the correct value of the KeyID field is used, an adversary can monitor other broadcast frames, or brute-force this value. The AP will then decrypt this packet using the group key. It will notice that the destination address (`addr3`) does not equal its own MAC address, and hence will forward the frame to the actual destination. If the destination MAC address is another wireless station, the AP will encrypt the frame using the appropriate pairwise key, and transmit it. As a result, the receiver will decrypt and process the forwarded, now unicast, frame. If the destination address is not a wireless station, it will be forwarded over the appropriate Ethernet connection. This technique can be used to inject IP packets, ARP packets, and so on, using the group key.

5.2 Decrypting All Traffic

Since unicast frames are encrypted with pairwise keys, we cannot directly use the group key to decrypt them. Nevertheless, it is possible to trick stations into sending all IP traffic to the broadcast MAC address, meaning the

group key will now be used to encrypt this traffic. This is done by performing an ARP poisoning attack. The malicious ARP packets are injected using the technique presented in Section 5.1. In our attack, we poison the ARP cache of the client so the IP address of the gateway is associated with the broadcast MAC address. Similarly, on the router, the IP address of the client will also be associated with a broadcast address. Since IP addresses in local networks are generally predictable, an attacker can brute-force the IP address of the client and router. After the ARP poisoning attack, all IP packets sent by the client and router are encrypted using the group key. An attacker can now capture and decrypt these packets. Furthermore, he can forward them to their real destination using the (unicast) packet injection technique of Section 5.1, so the victim will not notice he is under attack.

5.3 Experimental Verification

We tested this attack against an Asus RT-AC51U and a laptop running Windows 7. The group key was obtained by exploiting the weak random number generator as discussed in Section 3.4.1. In order to successfully perform the ARP poisoning attack against Windows, we injected malicious ARP requests. First, we were able to successfully inject the ARP packets using the group key. This confirms that the group key can be used to inject unicast packets. Once we poisoned the ARP cache of both the victim and router, they transmitted all their packets towards the broadcast MAC address. At this point we were able to successfully decrypt these broadcast packets using the group key, and read out the unicast IP packets sent by both the victim and router.

5.4 Countermeasures

If the network is operating in infrastructure mode, the AP should ignore all frames with a broadcast or multicast receiver address. This prevents an attacker from abusing the AP to forward unicast frames to stations. Another option is to disable all group traffic. While this may seem drastic, it is useful for protected but public hotspots. In these environments, connected stations do not trust each other, meaning group keys should not be used at all. Interestingly, the upcoming Hotspot 2.0 standard already supports this feature under the Downstream Group Addressed Forwarding (DGAF) option [49]. If DGAF is disabled, no group keys are configured, meaning the stations and AP ignore all group addressed Wi-Fi frames.

6 A New RNG for 802.11 Platforms

In this section we propose a random number generator that extracts randomness from fine-grained Received Signal Strength Indicator (RSSI) values. Specifically, we rely on the spectral scan feature of commodity 802.11 radios. This gives us roughly three million RSSI measurements per second, even if there is no background traffic.

6.1 Spectral Scan Feature

Most Atheros Wi-Fi radios, such as the AR9280, can perform RSSI measurements over the 56 sub-carriers used in high throughput (HT) OFDM[4]. Atheros calls this feature a spectral scan. It matches the requirement to decode HT OFDM modulated frames, where the channel is divided into 64 subcarriers [21, §18]. Eight of these subcarriers are used as guards to avoid channel cross talk, and are thus not sampled, resulting in 56 usable subcarriers. Therefore the spectral scan feature matches the normal OFDM demodulation requirements, and should be straightforward to implement by other vendors as well. The sweep time of one sample, i.e., spectral scan, is 4µs, and these scans can be made even when there is no background traffic. Each RSSI measurement is reported as an 8-bit value. After some optimazations, we could make our AR2980 chip generate around 50k samples per second. Since each sample contains 56 RSSI values, this totals to roughly three million measurements per second.

6.2 Random Number Generation

In our random number generator, we want to extract randomness out of every single RSSI measurement. Since our commodity devices can generate a large number of measurements per second, even when there is no background traffic, we need a fast method to process all these measurements. Hence our main goal is to design a technique to rapidly process all measurements. The resulting output can then be given as input to a system that properly extracts and manages randomness (see for example Yarrow-160 [25], or the model by Barak and Halevi [3] and its improvements [9]). In other words, our goal is only to design a method to rapidly process RSSI measurements which can be implemented in Wi-Fi radios, and to asses the quality of the resulting output.

We start by deriving one (possibly biased) bit out of each RSSI measurement. Any biases will be suppressed in a later step. Due to random variations in the background noise, the transmissions of other stations, and internal imperfections of the hardware, antenna, and radio, we expect that each RSSI measurement contains some amount of randomness. More concretely, we expect that the least significant bit of each RSSI measurement displays the most amount of randomness. To also take into account the other bits, we perform an exclusive or over all bits in the 8-bit RSSI measurement. Even if the other

[4] See `http://wikidevi.com/wiki/Atheros` for a list.

Table 2: Average number statistical test results for various configurations of the random number generator.

Configuration	Pass	Poor	Weak	Fail
1 bit per subcarrier	97.2%	0%	2.8%	0%
1 bit per spectral scan	98.1%	0%	1.9%	0%
normal mode	98.1%	0%	1.9%	0%

bits are not random, this can only increase the overall randomness. Since we are extracting 1 bit per subcarrier, we call this initial generator the "1 bit per subcarrier" configuration. When running the Dieharder statistical test suite [7] on this configuration, we noticed promising results (see Table 2). The Dieharder suite is a reimplementation of the Diehard tests [31], and in addition contains several tests from the NIST test suite [40]. While none of the tests fail, on average 2.8% of the tests return a weak result. However, this only means that the generated bits do not contain any obvious deficiencies. Subtle or small biases may still be present, and have to be filtered out. We do this by relying on the large number of measurements that our commodity devices can generate.

To suppress possible biases in the 1-bit per subcarrier construction, we combine several bits using an exclusive-or chain. More formally, if we have a sequence of bits $b_1, b_2, \ldots, b_n$, the exclusive-or chain of these bits is $bit = b_1 \oplus b_2 \oplus \ldots \oplus b_n$. Assuming that each bit b_i is equal to one with probability p, combining n bits in this manner has the following characteristics [10]:

$$\Pr[bit = 1] = 0.5 - 2^{n-1} \cdot (p - 0.5)^n \tag{2}$$

$$\Pr[bit = 0] = 0.5 + 2^{n-1} \cdot (p - 0.5)^n \tag{3}$$

We can now see that as n goes to infinity, both probabilities approach 0.5, meaning any possible biases will be suppressed. Moreover, an exclusive-or chain should also be straightforward to implement in a Wi-Fi radio.

In the second version of our generator, we use the exclusive-or chain to generate one random bit for each spectral scan sample. That is, all 56 random bits extracted from the subcarriers are XOR'ed together. We call this the "1-bit per spectral scan" mode. The reasoning behind this construction is that one bit is now influenced by all subcarriers, i.e., all available frequencies. An attacker that wants to influence the generation of any bit, now has to predict or influence all 56 subcarriers. The new results of the Dieharder tests show this improves the quality of the random numbers (see Table 2).

In a last step we combine 16 bits generated using the 1-bit per spectral scan construction. Again this is done using an exclusive-or chain. We call this the "normal mode". Interestingly, the results of the Dieharded test suite no longer improve. We conjecture that the 1.9% of tests that are marked as weak are statistical flukes: even a random steam of bits can look non-random at times.

Finally, we remark that in the normal execution mode of the generator, in total $56 \cdot 16$ bits are XOR'ed together. Hence, even if an attacker can correctly predict the 1 bit per subcarrier with a probability of 98%, our normal execution mode still outputs one bit that is close to uniform. More precisely, by relying on equation 2 and 3, the returned bit equals one with a probability of approximately $0.5 \cdot (1 - 2^{-52})$. Hence, by relying on the large number of measurements returned by the radio chip, even a very powerful attacker is unlikely to predict the final output of the generator. Furthermore, these bits are outputted at a speed of roughly 3125 bits per second. Finally, we believe that our technique can be efficiently implemented in Wi-Fi chips themselves. In practice, implementations can then query the Wi-Fi chip for random samples, and properly and securely manage this collected randomness using a model such as the one proposed by Barak and Halevi [3], or one of its improvements [9].

7 Related Work

While the random number generators that are used in certain browsers [14], OpenSSL [8], Linux [3, 16], GNU Privacy Guard [35], FreeBSD [50], and so on, have been widely studied, we are not aware of any works that study the random number generator of 802.11. More closely related to our work, Lorente et al. discovered that many routers generate weak, and sometimes predictable, default WPA2 passwords [27]. However, the random number generator of 802.11 is not used for this purpose, and hence was not analyzed.

The security of the 4-way handshake has been studied in several works [17, 18, 34]. These works revealed denial-of-service vulnerabilities [17, 34], or proof the security of an improved design [18]. Additionally, they focus on whether an attacker can perform a downgrade attack against the cipher used to protect traffic transmitted after the handshake. In contrast, we study downgrade attacks against the ciphers used to protect the handshake itself. The 802.11 standard also contains an informative analysis of the handshake [21, 11.6.6.8].

Many researchers have studied RC4 and its usage. Key recovery attacks against WEP were discovered [11], and were later improved in other works [48, 44, 43, 42]. In particular, Mantin and Klein studied whether the WEP key can still be recovered if the initial 256 bytes of RC4 are dropped [28, 26]. We extend this analysis by studying the impact of 16-byte initialization vectors as used in the 4-way handshake, and perform simulations of resulting attacks. AlFardan et al. showed that the initial 256 bytes of RC4 are biased [2]. Vanhoef and Piessens extended this result and showed that bytes between position 256

and 512 are also biased. [47]. In [6] Bricout et al. analyze the structure and exploitation of Mantin's *ABSAB* bias.

Security of group keys, and the isolation between unicast and group traffic, is briefly mentioned in the Hole 196 vulnerability [1]. However, this attack assumes that an associated (trusted) client will abuse the group key. Therefore it can only be considered an insider threat. Furthermore, it does not discuss how to inject unicast traffic using the group key, nor does it show how all internet traffic can be decrypted using the group key.

Several previous works use the RSSI measurements of 802.11 frames, as returned by commodity Wi-Fi radios, to create secret key agreement protocols [32, 23, 38]. Such a protocol negotiates a shared secret between two stations, that is unpredictable by observers. These works use the average RSSI over all subcarriers, meaning some entropy is lost compared to our per-subcarrier measurements. We use the spectral scan feature to perform RSSI measurements, which makes it possible to generate these measurements even if there is no background traffic. Models to properly collect and manage randomness, such as those contained in RSSI measurements, have also been studied. Examples are Yarrow-160 [25], the model by Barak and Halevi [3], or the model by Dodis et al. [9].

8 Conclusion

Although the generation of pairwise 802.11 keys has been widely analyzed, we have shown the same is not true for group keys. For certain devices the group key is easily predictable, which is caused by the faulty random number generator proposed in the 802.11 standard. This is especially problematic for Wi-Fi stacks in embedded devices, as they generally do not have other (standardized) sources of randomness. Furthermore, we have demonstrated a downgrade attack against the 4-way handshake, resulting in the usage of RC4 to protect the group key. An adversary can abuse this in an attempt to recover the group key.

We also showed that the group key can be used to inject any type of packet, and can even be used decrypt all internet traffic in a network. Combined with the faulty 802.11 random number generator, this enables an adversary to easily bypass both WPA-TKIP and AES-CCMP. To mitigate some of these issues, we also proposed and implemented a strong random number generator tailored for 802.11 platforms.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven. Mathy Vanhoef holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

References

[1] AHMAD, M. S. Wpa too! In *DEF CON* (2010).

[2] ALFARDAN, N. J., BERNSTEIN, D. J., PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. N. On the security of RC4 in TLS and WPA. In *USENIX Security* (2013).

[3] BARAK, B., AND HALEVI, S. A model and architecture for pseudo-random generation with applications to/dev/random. In *CCS* (2005).

[4] BITTAU, A., HANDLEY, M., AND LACKEY, J. The final nail in WEP's coffin. In *IEEE SP* (2006).

[5] BRADEN, R. Requirements for internet hosts – communication layers. RFC 1122, 1989.

[6] BRICOUT, R., MURPHY, S., PATERSON, K. G., AND VAN DER MERWE, T. Analysing and exploiting the mantin biases in RC4. Cryptology ePrint Archive, Report 2016/063, 2016.

[7] BROWN, R. G. Dieharder: A random number test suite. Available from `http://www.phy.duke.edu/~rgb/General/dieharder.php`, Feb. 2016.

[8] CHECKOWAY, S., NIEDERHAGEN, R., EVERSPAUGH, A., GREEN, M., LANGE, T., RISTENPART, T., BERNSTEIN, D. J., MASKIEWICZ, J., SHACHAM, H., AND FREDRIKSON, M. On the practical exploitability of Dual EC in TLS implementations. In *USENIX Security* (2014).

[9] DODIS, Y., POINTCHEVAL, D., RUHAULT, S., VERGNIAUD, D., AND WICHS, D. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 647–658.

[10] FAIRFIELD, R., MORTENSON, R., AND COULTHART, K. An LSI random number generator. In *CRYPTO* (1984).

[11] FLUHRER, S., MANTIN, I., AND SHAMIR, A. Weaknesses in the key scheduling algorithm of RC4. In *SAC* (2001).

[12] FLUHRER, S. R., AND MCGREW, D. A. Statistical analysis of the alleged RC4 keystream generator. In *FSE* (2000).

[13] FOUQUE, P.-A., MARTINET, G., VALETTE, F., AND ZIMMER, S. On the security of the CCM encryption mode and of a slight variant. In *Applied Cryptography and Network Security* (2008).

[14] GOLDBERG, I., AND WAGNER, D. Randomness and the netscape browser. *Dr. Dobb's Journal* (1996).

[15] GUPTA, S. S., MAITRA, S., MEIER, W., PAUL, G., AND SARKAR, S. Dependence in IV-related bytes of RC4 key enhances vulnerabilities in WPA. Cryptology ePrint Archive, Report 2013/476, 2013.

[16] GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. Analysis of the linux random number generator. In *IEEE SP* (2006).

[17] HE, C., AND MITCHELL, J. C. Analysis of the 802.1 i 4-Way handshake. In *WiSe* (2004), ACM.

[18] HE, C., SUNDARARAJAN, M., DATTA, A., DEREK, A., AND MITCHELL, J. C. A modular correctness proof of IEEE 802.11i and TLS. In *CCS* (2005).

[19] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security* (2012).

[20] IEEE. Motions to address some letter ballot 52 comments. In *802.11 WLANs WG proceedings* (2003).

[21] IEEE STD 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2012.

[22] IEEE STD 802.11I. *Amendment 6: Medium Access Control (MAC) Security Enhancements*, 2004.

[23] JANA, S., PREMNATH, S., CLARK, M., KASERA, S., PATWARI, N., AND KRISHNAMURTHY, S. On the effectiveness of secret key extraction from wireless signal strength. In *MobiCom* (2009).

[24] JONSSON, J. On the security of CTR+ CBC-MAC. In *SAC* (2002).

[25] KELSEY, J., SCHNEIER, B., AND FERGUSON, N. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography* (1999), Springer, pp. 13–33.

[26] KLEIN, A. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography* (2008).

[27] LORENTE, E. N., MEIJER, C., AND VERDULT, R. Scrutinizing WPA2 password generating algorithms in wireless routers. In *USENIX WOOT* (2015).

[28] MANTIN, I. A practical attack on the fixed RC4 in the WEP mode. In *AsiaCrypt* (2005).

[29] MANTIN, I. Predicting and distinguishing attacks on RC4 keystream generator. In *EUROCRYPT* (2005).

[30] MANTIN, I., AND SHAMIR, A. A practical attack on broadcast RC4. In *FSE* (2001).

[31] MARSAGLIA, G. Diehard tests of randomness. `http://stat.fsu.edu/pub/diehard/`.

[32] MATHUR, S., TRAPPE, W., MANDAYAM, N., YE, C., AND REZNIK, A. Radio-telepathy: extracting a secret key from an unauthenticated wireless channel. In *MobiCom* (2008).

[33] MIRONOV, I. (Not so) random shuffles of RC4. In *CRYPTO* (2002).

[34] MITCHELL, C. H. J. C. Security analysis and improvements for IEEE 802.11i. In *NDSS* (2005).

[35] NGUYEN, P. Q. Can we trust cryptographic software? cryptographic flaws in GNU privacy guard v1.2.3. In *EUROCRYPT* (2004).

[36] PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. Big bias hunting in amazonia: Large-scale computation and exploitation of RC4 biases. In *AsiaCrypt* (2014).

[37] PATERSON, K. G., SCHULDT, J. C. N., AND POETTERING, B. Plaintext recovery attacks against WPA/TKIP. In *FSE* (2014).

[38] PATWARI, N., CROFT, J., JANA, S., AND KASERA, S. High-rate uncorrelated bit extraction for shared secret key generation from channel measurements. *TMC* (2010).

[39] ROGAWAY, P., AND WAGNER, D. A critique of CCM. Cryptology ePrint Archive, Report 2003/070, 2003.

[40] RUKHIN, A., SOTO, J., NECHVATAL, J., SMID, M., AND BARKER, E. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Tech. rep., DTIC Document, 2001.

[41] SEPEHRDAD, P., SUSIL, P., VAUDENAY, S., AND VUAGNOUX, M. Tornado attack on RC4 with applications to WEP & WPA. Cryptology ePrint Archive, Report 2015/254, 2015.

[42] STUBBLEFIELD, A., IOANNIDIS, J., AND RUBIN, A. D. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *TISSEC* (2004).

[43] TEWS, E., AND BECK, M. Practical attacks against WEP and WPA. In *WiSec* (2009).

[44] TEWS, E., WEINMANN, R.-P., AND PYSHKIN, A. Breaking 104 bit WEP in less than 60 seconds. In *JISA*. 2007.

[45] VANHOEF, M., AND PIESSENS, F. Practical verification of WPA-TKIP vulnerabilities. In *ASIA CCS* (2013), ACM, pp. 427–436.

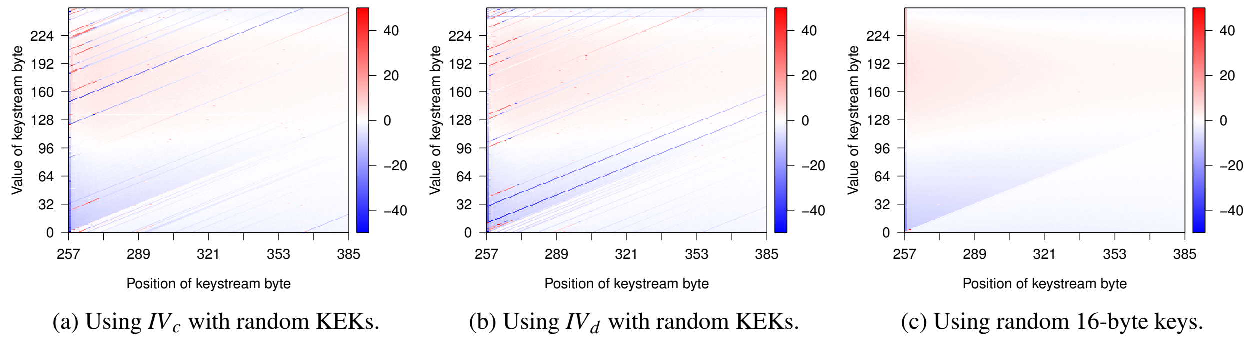
[46] VANHOEF, M., AND PIESSENS, F. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC* (2014).

[47] VANHOEF, M., AND PIESSENS, F. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security* (2015).

[48] VAUDENAY, S., AND VUAGNOUX, M. Passive–only key recovery attacks on RC4. In *SAC* (2007).

[49] WI-FI ALLIANCE. *Hotspot 2.0 (Release 2) Technical Specification v1.1.0*, 2010.

[50] WOOLLEY, R., MURRAY, M., DOUNIN, M., AND ERMILOV, R. arc4random(9): predictable sequence vulnerability.



(a) Using IV_c with random KEKs.

(b) Using IV_d with random KEKs.

(c) Using random 16-byte keys.

Figure 9: Biases in the RC4 keystream when concatenating a fixed 16-byte IV with a random 16-byte key (here called KEK key), and when using random 16-byte keys. Each points encodes a bias as the number $(pr - 2^{-8}) \cdot 2^{24}$, capped to values in $[-50, 50]$, with pr the empirical probability of the keystream byte value (y-axis) at a given location (x-axis).