All Your Biases Belong To Us: Breaking RC4 in WPA-TKIP and TLS

Mathy Vanhoef
KU Leuven
Mathy.Vanhoef@cs.kuleuven.be

Frank Piessens
KU Leuven
Frank.Piessens@cs.kuleuven.be

Abstract

We present new biases in RC4, break the Wi-Fi Protected Access Temporal Key Integrity Protocol (WPA-TKIP), and design a practical plaintext recovery attack against the Transport Layer Security (TLS) protocol. To empirically find new biases in the RC4 keystream we use statistical hypothesis tests. This reveals many new biases in the initial keystream bytes, as well as several new long-term biases. Our fixed-plaintext recovery algorithms are capable of using multiple types of biases, and return a list of plaintext candidates in decreasing likelihood.

To break WPA-TKIP we introduce a method to generate a large number of identical packets. This packet is decrypted by generating its plaintext candidate list, and using redundant packet structure to prune bad candidates. From the decrypted packet we derive the TKIP MIC key, which can be used to inject and decrypt packets. In practice the attack can be executed within an hour. We also attack TLS as used by HTTPS, where we show how to decrypt a secure cookie with a success rate of 94% using $9 \cdot 2^{27}$ ciphertexts. This is done by injecting known data around the cookie, abusing this using Mantin's *ABSAB* bias, and brute-forcing the cookie by traversing the plaintext candidates. Using our traffic generation technique, we are able to execute the attack in merely 75 hours.

1 Introduction

RC4 is (still) one of the most widely used stream ciphers. Arguably its most well known usage is in SSL and WEP, and in their successors TLS [8] and WPA-TKIP [19]. In particular it was heavily used after attacks against CBC-mode encryption schemes in TLS were published, such as BEAST [9], Lucky 13 [1], and the padding oracle attack [7]. As a mitigation RC4 was recommended. Hence, at one point around 50% of all TLS connections were using RC4 [2], with the current estimate around 30% [18]. This motivated the search for new attacks, relevant examples being [2, 20, 31, 15, 30]. Of special interest is the attack proposed by AlFardan et al., where roughly $13 \cdot 2^{30}$ ciphertexts are required to decrypt a cookie sent over HTTPS [2]. This corresponds to about 2000 hours of data in their setup, hence the attack is considered close to being practical. Our goal is to see how far these attacks can be pushed by exploring three areas. First, we search for new biases in the keystream. Second, we improve fixed-plaintext recovery algorithms. Third, we demonstrate techniques to perform our attacks in practice.

First we empirically search for biases in the keystream. This is done by generating a large amount of keystream, and storing statistics about them in several datasets. The resulting datasets are then analysed using statistical hypothesis tests. Our null hypothesis is that a keystream byte is uniformly distributed, or that two bytes are independent. Rejecting the null hypothesis is equivalent to detecting a bias. Compared to manually inspecting graphs, this allows for a more large-scale analysis. With this approach we found many new biases in the initial keystream bytes, as well as several new long-term biases.

We break WPA-TKIP by decrypting a complete packet using RC4 biases and deriving the TKIP MIC key. This key can be used to inject and decrypt packets [48]. In particular we modify the plaintext recovery attack of Paterson et al. [31, 30] to return a list of candidates in decreasing likelihood. Bad candidates are detected and pruned based on the (decrypted) CRC of the packet. This increases the success rate of simultaneously decrypting all unknown bytes. We achieve practicality using a novel method to rapidly inject identical packets into a network. In practice the attack can be executed within an hour.

We also attack RC4 as used in TLS and HTTPS, where we decrypt a secure cookie in realistic conditions. This is done by combining the *ABSAB* and Fluhrer-McGrew biases using variants of the of Isobe et al. and AlFardan et al. attack [20, 2]. Our technique can easily be extended to include other biases as well. To abuse Mantin's *ABSAB* bias we inject known plaintext around the cookie, and exploit this to calculate Bayesian plaintext likelihoods over

the unknown cookie. We then generate a list of (cookie) candidates in decreasing likelihood, and use this to brute-force the cookie in negligible time. The algorithm to generate candidates differs from the WPA-TKIP one due to the reliance on double-byte instead of single-byte likelihoods. All combined, we need $9 \cdot 2^{27}$ encryptions of a cookie to decrypt it with a success rate of 94%. Finally we show how to make a victim generate this amount within only 75 hours, and execute the attack in practice.

To summarize, our main contributions are:

- We use statistical tests to empirically detect biases in the keystream, revealing large sets of new biases.
- We design plaintext recovery algorithms capable of using multiple types of biases, which return a list of plaintext candidates in decreasing likelihood.
- We demonstrate practical exploitation techniques to break RC4 in both WPA-TKIP and TLS.

The remainder of this paper is organized as follows. Section 2 gives a background on RC4, TKIP, and TLS. In Sect. 3 we introduce hypothesis tests and report new biases. Plaintext recovery techniques are given in Sect. 4. Practical attacks on TKIP and TLS are presented in Sect. 5 and Sect. 6, respectively. Finally, we summarize related work in Sect. 7 and conclude in Sect. 8.

2 Background

We introduce RC4 and its usage in TLS and WPA-TKIP.

2.1 The RC4 Algorithm

The RC4 algorithm is intriguingly short and known to be very fast in software. It consists of a Key Scheduling Algorithm (KSA) and a Pseudo Random Generation Algorithm (PRGA), which are both shown in Fig. 1. The state consists of a permutation $\mathcal{S}$ of the set $\{0,\ldots,255\}$, a public counter i, and a private index j. The KSA takes as input a variable-length key and initializes $\mathcal{S}$. At each round $r = 1,2,\ldots$ of the PRGA, the yield statement outputs a keystream byte Z_r. All additions are performed modulo 256. A plaintext byte P_r is encrypted to ciphertext byte C_r using $C_r = P_r \oplus Z_r$.

2.1.1 Short-Term Biases

Several biases have been found in the initial RC4 keystream bytes. We call these short-term biases. The most significant one was found by Mantin and Shamir. They showed that the second keystream byte is twice as likely to be zero compared to uniform [25]. Or more formally that $\Pr[Z_2 = 0] \approx 2 \cdot 2^{-8}$, where the probability is over the random choice of the key. Because zero occurs more often than expected, we call this a positive bias. Similarly, a value occurring less often than expected is called a negative bias. This result was extended by Maitra et al. [23] and further refined by Sen Gupta et al. [38] to show that there is a bias towards zero for most initial keystream bytes. Sen Gupta et al. also found key-length dependent biases: if ℓ is the key length, keystream byte Z_ℓ has a positive bias towards $256 - \ell$ [38]. AlFardan et al. showed that all initial 256 keystream bytes are biased by empirically estimating their probabilities when 16-byte keys are used [2]. While doing this they found additional strong biases, an example being the bias towards value r for all positions $1 \leq r \leq 256$. This bias was also independently discovered by Isobe et al. [20].

The bias $\Pr[Z_1 = Z_2] = 2^{-8}(1 - 2^{-8})$ was found by Paul and Preneel [33]. Isobe et al. refined this result for the value zero to $\Pr[Z_1 = Z_2 = 0] \approx 3 \cdot 2^{-16}$ [20]. In [20] the authors searched for biases of similar strength between initial bytes, but did not find additional ones. However, we did manage to find new ones (see Sect. 3.3).

Listing (1) RC4 Key Scheduling (KSA).

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

Listing (2) RC4 Keystream Generation (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 1: Implementation of RC4 in Python-like pseudo-code. All additions are performed modulo 256.

2.1.2 Long-Term Biases

In contrast to short-term biases, which occur only in the initial keystream bytes, 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 digraphs, i.e., consecutive keystream bytes (Z_r, Z_{r+1}), deviate from uniform throughout the whole keystream [13]. These biases depend on the public counter i of the PRGA, and are listed in Table 1 (ignoring the condition on r for now). In their analysis, Fluhrer and McGrew assumed that the internal state of the RC4 algorithm was uniformly random.

Digraph	Condition	Probability
(0,0)	$i = 1$	$2^{-16}(1+2^{-7})$
(0,0)	$i \neq 1, 255$	$2^{-16}(1+2^{-8})$
(0,1)	$i \neq 0, 1$	$2^{-16}(1+2^{-8})$
(0,$i+1$)	$i \neq 0, 255$	$2^{-16}(1-2^{-8})$
($i+1$,255)	$i \neq 254 \wedge r \neq 1$	$2^{-16}(1+2^{-8})$
(129,129)	$i = 2, r \neq 2$	$2^{-16}(1+2^{-8})$
(255,$i+1$)	$i \neq 1, 254$	$2^{-16}(1+2^{-8})$
(255,$i+2$)	$i \in [1, 252] \wedge r \neq 2$	$2^{-16}(1+2^{-8})$
(255,0)	$i = 254$	$2^{-16}(1+2^{-8})$
(255,1)	$i = 255$	$2^{-16}(1+2^{-8})$
(255,2)	$i = 0, 1$	$2^{-16}(1+2^{-8})$
(255,255)	$i \neq 254 \wedge r \neq 5$	$2^{-16}(1-2^{-8})$

Table 1: Generalized Fluhrer-McGrew (FM) biases. Here i is the public counter in the PRGA and r the position of the first byte of the digraph. Probabilities for long-term biases are shown (for short-term biases see Fig. 4).

This assumption is only true after a few rounds of the PRGA [13, 26, 38]. Consequently these biases were generally not expected to be present in the initial keystream bytes. However, in Sect. 3.3.1 we show that most of these biases do occur in the initial keystream bytes, albeit with different probabilities than their long-term variants.

Another long-term bias was found by Mantin [24]. He discovered a 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. With the length of the gap $\mathcal{S}$ denoted by g, the bias can be written as:

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

Hence the bigger the gap, the weaker the bias. Finally, Sen Gupta et al. found the long-term bias [38]

$$\Pr[(Z_{w256}, Z_{w256+2}) = (0,0)] = 2^{-16}(1+2^{-8})$$

where $w \geq 1$. We discovered that a bias towards $(128, 0)$ is also present at these positions (see Sect. 3.4).

2.2 TKIP Cryptographic Encapsulation

The design goal of WPA-TKIP was for it to be a temporary replacement of WEP [19, §11.4.2]. While it is being phased out by the WiFi Alliance, a recent study shows its usage is still widespread [48]. Out of 6803 networks, they found that 71% of protected networks still allow TKIP, with 19% exclusively supporting TKIP.

Our attack on TKIP relies on two elements of the protocol: its weak Message Integrity Check (MIC) [44, 48], and its faulty per-packet key construction [2, 15, 31, 30]. We briefly introduce both aspects, assuming a 512-bit

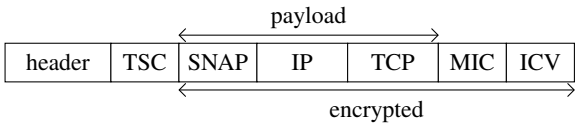

Figure 2: Simplified TKIP frame with a TCP payload.

Pairwise Transient Key (PTK) has already been negotiated between the Access Point (AP) and client. From this PTK a 128-bit temporal encryption key (TK) and two 64-bit Message Integrity Check (MIC) keys are derived. The first MIC key is used for AP-to-client communication, and the second for the reverse direction. Some works claim that the PTK, and its derived keys, are renewed after a user-defined interval, commonly set to 1 hour [44, 48]. However, we found that generally only the Groupwise Transient Key (GTK) is periodically renewed. Interestingly, our attack can be executed within an hour, so even networks which renew the PTK every hour can be attacked.

When the client wants to transmit a payload, it first calculates a MIC value using the appropriate MIC key and the Micheal algorithm (see Fig. Figure 2). Unfortunately Micheal is straightforward to invert: given plaintext data and its MIC value, we can efficiently derive the MIC key [44]. After appending the MIC value, a CRC checksum called the Integrity Check Value (ICV) is also appended. The resulting packet, including MAC header and example TCP payload, is shown in Figure 2. The payload, MIC, and ICV are encrypted using RC4 with a per-packet key. This key is calculated by a mixing function that takes as input the TK, the TKIP sequence counter (TSC), and the transmitter MAC address (TA). We write this as $\mathrm{K} = \mathrm{KM}(\mathrm{TA}, \mathrm{TK}, \mathrm{TSC})$. The TSC is a 6-byte counter that is incremented after transmitting a packet, and is included unencrypted in the MAC header. In practice the output of KM can be modelled as uniformly random [2, 31]. In an attempt to avoid weak-key attacks that broke WEP [12], the first three bytes of K are set to [19, §11.4.2.1.1]:

$$\mathrm{K}_0 = \mathrm{TSC}_1 \quad \mathrm{K}_1 = (\mathrm{TSC}_1 \mid 0\mathrm{x}20) \,\&\, 0\mathrm{x}7\mathrm{f} \quad \mathrm{K}_2 = \mathrm{TSC}_0$$

Here, TSC_0 and TSC_1 are the two least significant bytes of the TSC. Since the TSC is public, so are the first three bytes of K. Both formally and using simulations, it has been shown this actually weakens security [2, 15, 31, 30].

2.3 The TLS Record Protocol

We focus on the TLS record protocol when RC4 is selected as the symmetric cipher [8]. In particular we assume the handshake phase is completed, and a 48-byte TLS master secret has been negotiated.

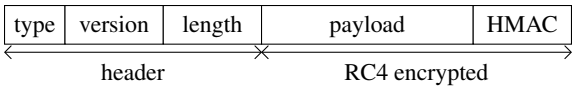

Figure 3: TLS Record structure when using RC4.

To send an encrypted payload, a TLS record of type application data is created. It contains the protocol version, length of the encrypted content, the payload itself, and finally an HMAC. The resulting layout is shown in Fig. 3. The HMAC is computed over the header, a sequence number incremented for each transmitted record, and the plaintext payload. Both the payload and HMAC are encrypted. At the start of a connection, RC4 is initialized with a key derived from the TLS master secret. This key can be modelled as being uniformly random [2]. None of the initial keystream bytes are discarded.

In the context of HTTPS, one TLS connection can be used to handle multiple HTTP requests. This is called a persistent connection. Slightly simplified, a server indicates support for this by setting the HTTP `Connection` header to `keep-alive`. This implies RC4 is initialized only once to send all HTTP requests, allowing the usage of long-term biases in attacks. Finally, cookies can be marked as being `secure`, assuring they are transmitted only over a TLS connection.

3 Empirically Finding New Biases

In this section we explain how to empirically yet soundly detect biases. While we discovered many biases, we will not use them in our attacks. This simplifies the description of the attacks. And, while using the new biases may improve our attacks, using existing ones already sufficed to significantly improve upon existing attacks. Hence our focus will mainly be on the most intriguing new biases.

3.1 Soundly Detecting Biases

In order to empirically detect new biases, we rely on hypothesis tests. That is, we generate keystream statistics over random RC4 keys, and use statistical tests to uncover deviations from uniform. This allows for a large-scale and automated analysis. To detect single-byte biases, our null hypothesis is that the keystream byte values are uniformly distributed. To detect biases between two bytes, one may be tempted to use as null hypothesis that the pair is uniformly distributed. However, this falls short if there are already single-byte biases present. In this case single-byte biases imply that the pair is also biased, while both bytes may in fact be independent. Hence, to detect double-byte biases, our null hypothesis is that they are independent. With this test, we even detected pairs that are actually more uniform than expected. Rejecting the null hypothesis is now the same as detecting a bias.

To test whether values are uniformly distributed, we use a chi-squared goodness-of-fit test. A naive approach to test whether two bytes are independent, is using a chi-squared independence test. Although this would work, it is not ideal when only a few biases (outliers) are present. Moreover, based on previous work we expect that only a few values between keystream bytes show a clear dependency on each other [13, 24, 20, 38, 4]. Taking the Fluhrer-McGrew biases as an example, at any position at most 8 out of a total 65536 value pairs show a clear bias [13]. When expecting only a few outliers, the M-test of Fuchs and Kenett can be asymptotically more powerful than the chi-squared test [14]. Hence we use the M-test to detect dependencies between keystream bytes. To determine which values are biased between dependent bytes, we perform proportion tests over all value pairs.

We reject the null hypothesis only if the p-value is lower than 10^{-4}. Holm's method is used to control the family-wise error rate when performing multiple hypothesis tests. This controls the probability of even a single false positive over all hypothesis tests. We always use the two-sided variant of an hypothesis test, since a bias can be either positive or negative.

Simply giving or plotting the probability of two dependent bytes is not ideal. After all, this probability includes the single-byte biases, while we only want to report the strength of the dependency between both bytes. To solve this, we report the absolute relative bias compared to the expected single-byte based probability. More precisely, say that by multiplying the two single-byte probabilities of a pair, we would expect it to occur with probability p. Given that this pair actually occurs with probability s, we then plot the value $|q|$ from the formula $s = p \cdot (1+q)$. In a sense the relative bias indicates how much information is gained by not just considering the single-byte biases, but using the real byte-pair probability.

3.2 Generating Datasets

In order to generate detailed statistics of keystream bytes, we created a distributed setup. We used roughly 80 standard desktop computers and three powerful servers as workers. The generation of the statistics is done in C. Python was used to manage the generated datasets and control all workers. On start-up each worker generates a cryptographically random AES key. Random 128-bit RC4 keys are derived from this key using AES in counter mode. Finally, we used R for all statistical analysis [34].

Our main results are based on two datasets, called `first16` and `consec512`. The `first16` dataset estimates $\Pr[Z_a = x \wedge Z_b = y]$ for $1 \leq a \leq 16$, $1 \leq b \leq 256$, and $0 \leq x, y < 256$ using 2^{44} keys. Its generation took

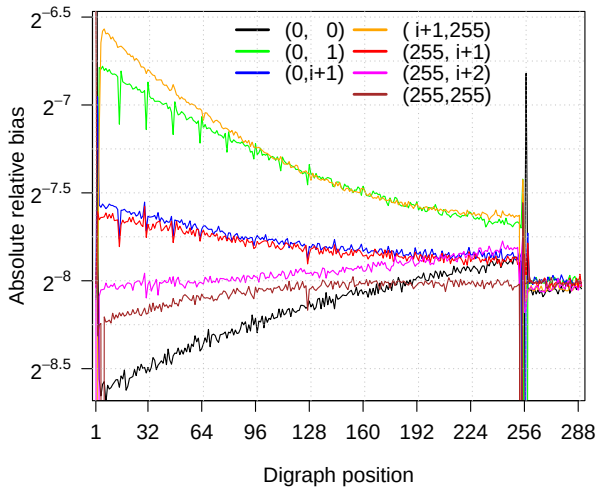

Figure 4: Absolute relative bias of several Fluhrer-McGrew digraphs in the initial keystream bytes, compared to their expected single-byte based probability.

roughly 9 CPU years. This allows detecting biases between the first 16 bytes and the other initial 256 bytes. The `consec512` dataset estimates $\Pr[Z_r = x \wedge Z_{r+1} = y]$ for $1 \leq r \leq 512$ and $0 \leq x, y < 256$ using 2^{45} keys, which took 16 CPU years to generate. It allows a detailed study of consecutive keystream bytes up to position 512.

We optimized the generation of both datasets. The first optimization is that one run of a worker generates at most 2^{30} keystreams. This allows usage of 16-bit integers for all counters collecting the statistics, even in the presence of significant biases. Only when combining the results of workers are larger integers required. This lowers memory usage, reducing cache misses. To further reduce cache misses we generate several keystreams before updating the counters. In independent work, Paterson et al. used similar optimizations [30]. For the `first16` dataset we used an additional optimization. Here we first generate several keystreams, and then update the counters in a sorted manner based on the value of Z_a. This optimization caused the most significant speed-up for the `first16` dataset.

3.3 New Short-Term Biases

By analysing the generated datasets we discovered many new short-term biases. We classify them into several sets.

3.3.1 Biases in (Non-)Consecutive Bytes

By analysing the `consec512` dataset we discovered numerous biases between consecutive keystream bytes. Our first observation is that the Fluhrer-McGrew biases are also present in the initial keystream bytes. Exceptions occur at positions 1, 2 and 5, and are listed in Table 1 (note the extra conditions on the position r). This is surprising, as the Fluhrer-McGrew biases were generally not expected to be present in the initial keystream bytes [13]. However, these biases are present, albeit with different probabilities. Figure 4 shows the absolute relative bias of most Fluhrer-McGrew digraphs, compared to their expected single-byte based probability (recall Sect. 3.1). For all digraphs, the sign of the relative bias q is the same as its long-term variant as listed in Table 1. We observe that the relative biases converge to their long-term values, especially after position 257. The vertical lines around position 1 and 256 are caused by digraphs which do not hold (or hold more strongly) around these positions.

First byte	Second byte	Probability
Consecutive biases:		
$Z_{15} = 240$	$Z_{16} = 240$	$2^{-15.94786}(1 - 2^{-4.894})$
$Z_{31} = 224$	$Z_{32} = 224$	$2^{-15.96486}(1 - 2^{-5.427})$
$Z_{47} = 208$	$Z_{48} = 208$	$2^{-15.97595}(1 - 2^{-5.963})$
$Z_{63} = 192$	$Z_{64} = 192$	$2^{-15.98363}(1 - 2^{-6.469})$
$Z_{79} = 176$	$Z_{80} = 176$	$2^{-15.99020}(1 - 2^{-7.150})$
$Z_{95} = 160$	$Z_{96} = 160$	$2^{-15.99405}(1 - 2^{-7.740})$
$Z_{111} = 144$	$Z_{112} = 144$	$2^{-15.99668}(1 - 2^{-8.331})$
Non-consecutive biases:		
$Z_3 = 4$	$Z_5 = 4$	$2^{-16.00243}(1 + 2^{-7.912})$
$Z_3 = 131$	$Z_{131} = 3$	$2^{-15.99543}(1 + 2^{-8.700})$
$Z_3 = 131$	$Z_{131} = 131$	$2^{-15.99347}(1 - 2^{-9.511})$
$Z_4 = 5$	$Z_6 = 255$	$2^{-15.99918}(1 + 2^{-8.208})$
$Z_{14} = 0$	$Z_{16} = 14$	$2^{-15.99349}(1 + 2^{-9.941})$
$Z_{15} = 47$	$Z_{17} = 16$	$2^{-16.00191}(1 + 2^{-11.279})$
$Z_{15} = 112$	$Z_{32} = 224$	$2^{-15.96637}(1 - 2^{-10.904})$
$Z_{15} = 159$	$Z_{32} = 224$	$2^{-15.96574}(1 + 2^{-9.493})$
$Z_{16} = 240$	$Z_{31} = 63$	$2^{-15.95021}(1 + 2^{-8.996})$
$Z_{16} = 240$	$Z_{32} = 16$	$2^{-15.94976}(1 + 2^{-9.261})$
$Z_{16} = 240$	$Z_{33} = 16$	$2^{-15.94960}(1 + 2^{-10.516})$
$Z_{16} = 240$	$Z_{40} = 32$	$2^{-15.94976}(1 + 2^{-10.933})$
$Z_{16} = 240$	$Z_{48} = 16$	$2^{-15.94989}(1 + 2^{-10.832})$
$Z_{16} = 240$	$Z_{48} = 208$	$2^{-15.92619}(1 - 2^{-10.965})$
$Z_{16} = 240$	$Z_{64} = 192$	$2^{-15.93357}(1 - 2^{-11.229})$

Table 2: Biases between (non-consecutive) bytes.

A second set of strong biases have the form:

$$\Pr[Z_{w16-1} = Z_{w16} = 256 - w16] \tag{2}$$

with $1 \leq w \leq 7$. In Table 2 we list their probabilities. Since 16 equals our key length, these are likely key-length dependent biases.

Another set of biases have the form $\Pr[Z_r = Z_{r+1} = x]$. Depending on the value x, these biases are either negative or positive. Hence summing over all x and calculating $\Pr[Z_r = Z_{r+1}]$ would lose some statistical informa-

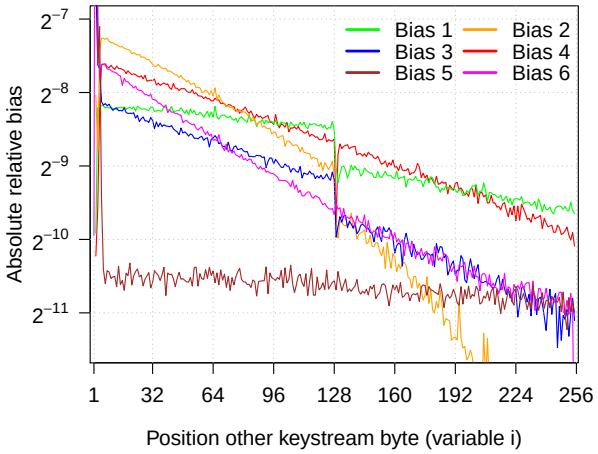

Figure 5: Biases induced by the first two bytes. The number of the biases correspond to those in Sect. 3.3.2.

tion. In principle, these biases also include the Fluhrer-McGrew pairs $(0,0)$ and $(255,255)$. However, as the bias for both these pairs is much higher than for other values, we don't include them here. Our new bias, in the form of $\Pr[Z_r = Z_{r+1}]$, was detected up to position 512.

We also detected biases between non-consecutive bytes that do not fall in any obvious categories. An overview of these is given in Table 2. We remark that the biases induced by $Z_{16} = 240$ generally have a position, or value, that is a multiple of 16. This is an indication that these are likely key-length dependent biases.

3.3.2 Influence of Z_1 and Z_2

Arguably our most intriguing finding is the amount of information the first two keystream bytes leak. In particular, Z_1 and Z_2 influence all initial 256 keystream bytes. We detected the following six sets of biases:

1) $Z_1 = 257 - i \wedge Z_i = 0$
2) $Z_1 = 257 - i \wedge Z_i = i$
3) $Z_1 = 257 - i \wedge Z_i = 257 - i$
4) $Z_1 = i - 1 \wedge Z_i = 1$
5) $Z_2 = 0 \wedge Z_i = 0$
6) $Z_2 = 0 \wedge Z_i = i$

Their absolute relative bias, compared to the single-byte biases, is shown in Fig. 5. The relative bias of pairs 5 and 6, i.e., those involving Z_2, are generally negative. Pairs involving Z_1 are generally positive, except pair 3, which always has a negative relative bias. We also detected dependencies between Z_1 and Z_2 other than the $\Pr[Z_1 = Z_2]$ bias of Paul and Preneel [33]. That is, the following pairs are strongly biased:

A) $Z_1 = 0 \wedge Z_2 = x$
B) $Z_1 = x \wedge Z_2 = 258 - x$
C) $Z_1 = x \wedge Z_2 = 0$
D) $Z_1 = x \wedge Z_2 = 1$

Bias A and C are negative for all $x \neq 0$, and both appear to be mainly caused by the strong positive bias

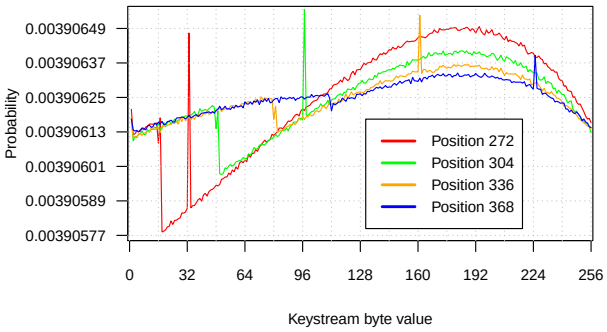

Figure 6: Single-byte biases beyond position 256.

$\Pr[Z_1 = Z_2 = 0]$ found by Isobe et al. Bias B and D are positive. We also discovered the following three biases:

$$\Pr[Z_1 = Z_3] = 2^{-8}(1 - 2^{-9.617}) \quad (3)$$
$$\Pr[Z_1 = Z_4] = 2^{-8}(1 + 2^{-8.590}) \quad (4)$$
$$\Pr[Z_2 = Z_4] = 2^{-8}(1 - 2^{-9.622}) \quad (5)$$

Note that all either involve an equality with Z_1 or Z_2.

3.3.3 Single-Byte Biases

We analysed single-byte biases by aggregating the `consec512` dataset, and by generating additional statistics specifically for single-byte probabilities. The aggregation corresponds to calculating

$$\Pr[Z_r = k] = \sum_{y=0}^{255} \Pr[Z_r = k \wedge Z_{r+1} = y] \quad (6)$$

We ended up with 2^{47} keys used to estimate single-byte probabilities. For all initial 513 bytes we could reject the hypothesis that they are uniformly distributed. In other words, all initial 513 bytes are biased. Figure 6 shows the probability distribution for some positions. Manual inspection of the distributions revealed a significant bias towards $Z_{256+k\cdot 16} = k \cdot 32$ for $1 \leq k \leq 7$. These are likely key-length dependent biases. Following [26] we conjecture there are single-byte biases even beyond these positions, albeit less strong.

3.4 New Long-Term Biases

To search for new long-term biases we created a variant of the `first16` dataset. It estimates

$$\Pr[Z_{256w+a} = x \wedge Z_{256w+b} = y] \quad (7)$$

for $0 \leq a \leq 16$, $0 \leq b < 256$, $0 \leq x, y < 256$, and $w \geq 4$. It is generated using 2^{12} RC4 keys, where each key was used to generate 2^{40} keystream bytes. This took roughly 8 CPU years. The condition on w means we always

dropped the initial 1023 keystream bytes. Using this dataset we can detect biases whose periodicity is a proper divisor of 256 (e.g., it detected all Fluhrer-McGrew biases). Our new short-term biases were not present in this dataset, indicating they indeed only occur in the initial keystream bytes, at least with the probabilities we listed. We did find the new long-term bias

$$\Pr[(Z_{w256}, Z_{w256+2}) = (128, 0)] = 2^{-16}(1 + 2^{-8}) \quad (8)$$

for $w \geq 1$. Surprisingly this was not discovered earlier, since a bias towards $(0,0)$ at these positions was already known [38]. We also specifically searched for biases of the form $\Pr[Z_r = Z_{r'}]$ by aggregating our dataset. This revealed that many bytes are dependent on each other. That is, we detected several long-term biases of the form

$$\Pr[Z_{256w+a} = Z_{256w+b}] \approx 2^{-8}(2 \pm 2^{-16}) \quad (9)$$

Due to the small relative bias of 2^{-16}, these are difficult to reliably detect. That is, the pattern where these biases occur, and when their relative bias is positive or negative, is not yet clear. We consider it an interesting future research direction to (precisely and reliably) detect all keystream bytes which are dependent in this manner.

4 Plaintext Recovery

We will design plaintext recovery techniques for usage in two areas: decrypting TKIP packets and HTTPS cookies. In other scenarios, variants of our methods can be used.

4.1 Calculating Likelihood Estimates

Our goal is to convert a sequence of ciphertexts $\mathcal{C}$ into predictions about the plaintext. This is done by exploiting biases in the keystream distributions $p_k = \Pr[Z_r = k]$. These can be obtained by following the steps in Sect. 3.2. All biases in p_k are used to calculate the likelihood that a plaintext byte equals a certain value μ. To accomplish this, we rely on the likelihood calculations of AlFardan et al. [2]. Their idea is to calculate, for each plaintext value μ, the (induced) keystream distributions required to witness the captured ciphertexts. The closer this matches the real keystream distributions p_k, the more likely we have the correct plaintext byte. Assuming a fixed position r for simplicity, the induced keystream distributions are defined by the vector $N^\mu = (N_0^\mu, \ldots, N_{255}^\mu)$. Each N_k^μ represents the number of times the keystream byte was equal to k, assuming the plaintext byte was μ:

$$N_k^\mu = |\{C \in \mathcal{C} \mid C = k \oplus \mu\}| \quad (10)$$

Note that the vectors N^μ and $N^{\mu'}$ are permutations of each other. Based on the real keystream probabilities p_k we calculate the likelihood that this induced distribution would occur in practice. This is modelled using a multinomial distribution with the number of trails equal to $|\mathcal{C}|$, and the categories being the 256 possible keystream byte values. Since we want the probability of this *sequence* of keystream bytes we get [30]:

$$\Pr[\mathcal{C} \mid P = \mu] = \prod_{k \in \{0,\ldots,255\}} (p_k)^{N_k^\mu} \quad (11)$$

Using Bayes' theorem we can convert this into the likelihood λ_μ that the plaintext byte is μ:

$$\lambda_\mu = \Pr[P = \mu \mid \mathcal{C}] \sim \Pr[\mathcal{C} \mid P = \mu] \quad (12)$$

For our purposes we can treat this as an equality [2]. The most likely plaintext byte μ is the one that maximises λ_μ. This was extended to a pair of dependent keystream bytes in the obvious way:

$$\lambda_{\mu_1,\mu_2} = \prod_{k_1,k_2 \in \{0,\ldots,255\}} (p_{k_1,k_2})^{N_{k_1,k_2}^{\mu_1,\mu_2}} \quad (13)$$

We found this formula can be optimized if most keystream byte values k_1 and k_2 are independent and uniform. More precisely, let us assume that all keystream value pairs in the set $\mathcal{I}$ are independent and uniform:

$$\forall (k_1,k_2) \in \mathcal{I}\colon\ p_{k_1,k_2} = p_{k_1} \cdot p_{k_2} = u \quad (14)$$

where u represents the probability of an unbiased double-byte keystream value. Then we rewrite formula 13 to:

$$\lambda_{\mu_1,\mu_2} = (u)^{M^{\mu_1,\mu_2}} \cdot \prod_{k_1,k_2 \in \mathcal{I}^c} (p_{k_1,k_2})^{N_{k_1,k_2}^{\mu_1,\mu_2}} \quad (15)$$

where

$$M^{\mu_1,\mu_2} = \sum_{k_1,k_2 \in \mathcal{I}} N_{k_1,k_2}^{\mu_1,\mu_2} = |\mathcal{C}| - \sum_{k_1,k_2 \in \mathcal{I}^c} N_{k_1,k_2}^{\mu_1,\mu_2} \quad (16)$$

and with $\mathcal{I}^c$ the set of dependent keystream values. If the set $\mathcal{I}^c$ is small, this results in a lower time-complexity. For example, when applied to the long-term keystream setting over Fluhrer-McGrew biases, roughly 2^{19} operations are required to calculate all likelihood estimates, instead of 2^{32}. A similar (though less drastic) optimization can also be made when single-byte biases are present.

4.2 Likelihoods From Mantin's Bias

We now show how to compute a double-byte plaintext likelihood using Mantin's *ABSAB* bias. More formally, we want to compute the likelihood λ_{μ_1,μ_2} that the plaintext bytes at fixed positions r and $r+1$ are μ_1 and μ_2, respectively. To accomplish this we abuse surrounding known plaintext. Our main idea is to first calculate the

likelihood of the *differential* between the known and unknown plaintext. We define the differential $\widehat{Z}_r^g$ as:

$$\widehat{Z}_r^g = (Z_r \oplus Z_{r+2+g}, Z_{r+1} \oplus Z_{r+3+g}) \tag{17}$$

Similarly we use $\widehat{C}_r^g$ and $\widehat{P}_r^g$ to denote the differential over ciphertext and plaintext bytes, respectively. The *ABSAB* bias can then be written as:

$$\Pr[\widehat{Z}_r^g = (0,0)] = 2^{-16}(1 + 2^{-8}\mathrm{e}^{\frac{-4-8g}{256}}) = \alpha(g) \tag{18}$$

When XORing both sides of $\widehat{Z}_r^g = (0,0)$ with $\widehat{P}_r^g$ we get

$$\Pr[\widehat{C}_r^g = \widehat{P}_r^g] = \alpha(g) \tag{19}$$

Hence Mantin's bias implies that the ciphertext differential is biased towards the plaintext differential. We use this to calculate the likelihood $\lambda_{\widehat{\mu}}$ of a differential $\widehat{\mu}$. For ease of notation we assume a fixed position r and a fixed *ABSAB* gap of g. Let $\widehat{\mathcal{C}}$ be the sequence of captured ciphertext differentials, and μ_1' and μ_2' the known plaintext bytes at positions $r+2+g$ and $r+3+g$, respectively. Similar to our previous likelihood estimates, we calculate the probability of witnessing the ciphertext differentials $\widehat{\mathcal{C}}$ assuming the plaintext differential is $\widehat{\mu}$:

$$\Pr[\widehat{\mathcal{C}} \mid \widehat{P} = \widehat{\mu}] = \prod_{\widehat{k} \in \{0,\ldots,255\}^2} \Pr[\widehat{Z} = \widehat{k}]^{N_{\widehat{k}}^{\widehat{\mu}}} \tag{20}$$

where

$$N_{\widehat{k}}^{\widehat{\mu}} = \left| \left\{ \widehat{C} \in \widehat{\mathcal{C}} \mid \widehat{C} = \widehat{k} \oplus \widehat{\mu} \right\} \right| \tag{21}$$

Using this notation we see that this is indeed identical to an ordinary likelihood estimation. Using Bayes' theorem we get $\lambda_{\widehat{\mu}} = \Pr[\widehat{\mathcal{C}} \mid \widehat{P} = \widehat{\mu}]$. Since only one differential pair is biased, we can apply and simplify formula 15:

$$\lambda_{\widehat{\mu}} = (1 - \alpha(g))^{|\mathcal{C}| - |\widehat{\mu}|} \cdot \alpha(g)^{|\widehat{\mu}|} \tag{22}$$

where we slightly abuse notation by defining $|\widehat{\mu}|$ as

$$|\widehat{\mu}| = \left| \left\{ \widehat{C} \in \widehat{\mathcal{C}} \mid \widehat{C} = \widehat{\mu} \right\} \right| \tag{23}$$

Finally we apply our knowledge of the known plaintext bytes to get our desired likelihood estimate:

$$\lambda_{\mu_1,\mu_2} = \lambda_{\widehat{\mu} \oplus (\mu_1', \mu_2')} \tag{24}$$

To estimate at which gap size the *ABSAB* bias is still detectable, we generated 2^{48} blocks of 512 keystream bytes. Based on this we empirically confirmed Mantin's *ABSAB* bias up to gap sizes of at least 135 bytes. The theoretical estimate in formula 1 slightly underestimates the true empirical bias. In our attacks we use a maximum gap size of 128.

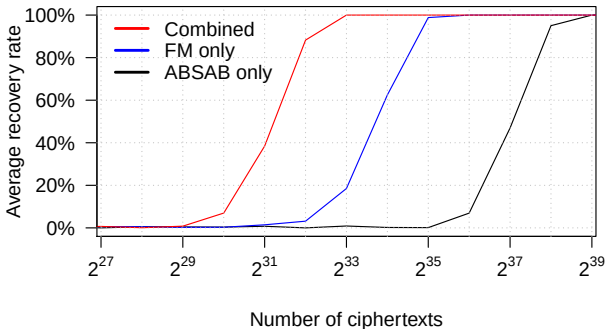

Figure 7: Average success rate of decrypting two bytes using: (1) one *ABSAB* bias; (2) Fluhrer-McGrew (FM) biases; and (3) combination of FM biases with 258 *ABSAB* biases. Results based on 2048 simulations each.

4.3 Combining Likelihood Estimates

Our goal is to combine multiple types of biases in a likelihood calculation. Unfortunately, if the biases cover overlapping positions, it quickly becomes infeasible to perform a single likelihood estimation over all bytes. In the worst case, the calculation cannot be optimized by relying on independent biases. Hence, a likelihood estimate over n keystream positions would have a time complexity of $\mathcal{O}(2^{2 \cdot 8 \cdot n})$. To overcome this problem, we perform and combine multiple separate likelihood estimates.

We will combine multiple types of biases by multiplying their individual likelihood estimates. For example, let λ'_{μ_1,μ_2} be the likelihood of plaintext bytes μ_1 and μ_2 based on the Fluhrer-McGrew biases. Similarly, let λ'_{g,μ_1,μ_2} be likelihoods derived from *ABSAB* biases of gap g. Then their combination is straightforward:

$$\lambda_{\mu_1,\mu_2} = \lambda'_{\mu_1,\mu_2} \cdot \prod_g \lambda'_{g,\mu_1,\mu_2} \tag{25}$$

While this method may not be optimal when combining likelihoods of dependent bytes, it does appear to be a general and powerful method. An open problem is determining which biases can be combined under a single likelihood calculation, while keeping computational requirements acceptable. Likelihoods based on other biases, e.g., Sen Gupta's and our new long-term biases, can be added as another factor (though some care is needed so positions properly overlap).

To verify the effectiveness of this approach, we performed simulations where we attempt to decrypt two bytes using one double-byte likelihood estimate. First this is done using only the Fluhrer-McGrew biases, and using only one *ABSAB* bias. Then we combine $2 \cdot 129$ *ABSAB* biases and the Fluhrer-McGrew biases, where we use the method from Sect. 4.2 to derive likelihoods from *ABSAB* biases. We assume the unknown bytes are surrounded at both sides by known plaintext, and use a

maximum *ABSAB* gap of 128 bytes. Figure 7 shows the results of this experiment. Notice that a single *ABSAB* bias is weaker than using all Fluhrer-McGrew biases at a specific position. However, combining several *ABSAB* biases clearly results in a major improvement. We conclude that our approach to combine biases significantly reduces the required number of ciphertexts.

4.4 List of Plaintext Candidates

In practice it is useful to have a list of plaintext candidates in decreasing likelihood. For example, by traversing this list we could attempt to brute-force keys, passwords, cookies, etc. (see Sect. 6). In other situations the plaintext may have a rigid structure allowing the removal of candidates (see Sect. 5). We will generate a list of plaintext candidates in decreasing likelihood, when given either single-byte or double-byte likelihood estimates.

First we show how to construct a candidate list when given single-byte plaintext likelihoods. While it is trivial to generate the two most likely candidates, beyond this point the computation becomes more tedious. Our solution is to incrementally compute the N most likely candidates based on their length. That is, we first compute the N most likely candidates of length 1, then of length 2, and so on. Algorithm 1 gives a high-level implementation of this idea. Variable $P_r[i]$ denotes the i-th most likely plaintext of length r, having a likelihood of $E_r[i]$. The two `min` operations are needed because in the initial loops we are not yet be able to generate N candidates, i.e., there only exist 256^r plaintexts of length r. Picking the μ' which maximizes $pr(\mu')$ can be done efficiently using a priority queue. In practice, only the latest two versions of lists E and P need to be stored. To better maintain numeric stability, and to make the computation more efficient, we perform calculations using the logarithm of the likelihoods. We implemented Algorithm 1 and report on its performance in Sect. 5, where we use it to attack a wireless network protected by WPA-TKIP.

To generate a list of candidates from double-byte likelihoods, we first show how to model the likelihoods as a hidden Markov model (HMM). This allows us to present a more intuitive version of our algorithm, and refer to the extensive research in this area if more efficient implementations are needed. The algorithm we present can be seen as a combination of the classical Viterbi algorithm, and Algorithm 1. Even though it is not the most optimal one, it still proved sufficient to significantly improve plaintext recovery (see Sect. 6). For an introduction to HMMs we refer the reader to [35]. Essentially an HMM models a system where the internal states are not observable, and after each state transition, output is (probabilistically) produced dependent on its new state.

We model the plaintext likelihood estimates as a first-

Algorithm 1: Generate plaintext candidates in decreasing likelihood using single-byte estimates.

Input: L : Length of the unknown plaintext
$\lambda_{1\le r\le L,\,0\le\mu\le 255}$: single-byte likelihoods
N: Number of candidates to generate
Returns: List of candidates in decreasing likelihood

$P_0[1] \leftarrow \varepsilon$
$E_0[1] \leftarrow 0$
for $r = 1$ **to** L **do**
  for $\mu = 0$ **to** 255 **do**
    $pos(\mu) \leftarrow 1$
    $pr(\mu) \leftarrow E_{r-1}[1] + \log(\lambda_{r,\mu})$
  for $i = 1$ **to** $\min(N, 256^r)$ **do**
    $\mu \leftarrow \mu'$ which maximizes $pr(\mu')$
    $P_r[i] \leftarrow P_{r-1}[pos(\mu)] \,\|\, \mu$
    $E_r[i] \leftarrow E_{r-1}[pos(\mu)] + \log(\lambda_{r,\mu})$
    $pos(\mu) \leftarrow pos(\mu) + 1$
    $pr(\mu) \leftarrow E_{r-1}[pos(\mu)] + \log(\lambda_{r,\mu})$
    if $pos(\mu) > \min(N, 256^{r-1})$ **then**
      $pr(\mu) \leftarrow -\infty$
return P_N

order time-inhomogeneous HMM. The state space S of the HMM is defined by the set of possible plaintext values $\{0,\ldots,255\}$. The byte positions are modelled using the time-dependent (i.e., inhomogeneous) state transition probabilities. Intuitively, the "current time" in the HMM corresponds to the current plaintext position. This means the transition probabilities for moving from one state to another, which normally depend on the current time, will now depend on the position of the byte. More formally:

$$\Pr[S_{t+1} = \mu_2 \mid S_t = \mu_1] \sim \lambda_{t,\mu_1,\mu_2} \tag{26}$$

where t represents the time. For our purposes we can treat this as an equality. In an HMM it is assumed that its current state is not observable. This corresponds to the fact that we do not know the value of any plaintext bytes. In an HMM there is also some form of output which depends on the current state. In our setting a particular plaintext value leaks no observable (side-channel) information. This is modelled by always letting every state produce the same null output with probability one.

Using the above HMM model, finding the most likely plaintext reduces to finding the most likely state sequence. This is solved using the well-known Viterbi algorithm. Indeed, the algorithm presented by AlFardan et al. closely resembles the Viterbi algorithm [2]. Similarly, finding the N most likely plaintexts is the same as finding the N most likely state sequences. Hence any N-best variant of the Viterbi algorithm (also called list Viterbi

Algorithm 2: Generate plaintext candidates in decreasing likelihood using double-byte estimates.

Input: L : Length of the unknown plaintext plus two
m_1 and m_L: known first and last byte
$\lambda_{1\le r<L,\,0\le\mu_1,\mu_2\le 255}$: double-byte likelihoods
N: Number of candidates to generate
Returns: List of candidates in decreasing likelihood

for $\mu_2 = 0$ **to** 255 **do**
  $E_2[\mu_2, 1] \leftarrow \log(\lambda_{1,m_1,\mu_2})$
  $P_2[\mu_2, 1] \leftarrow m_1 \,||\, \mu_2$

for $r = 3$ **to** L **do**
  for $\mu_2 = 0$ **to** 255 **do**
    for $\mu_1 = 0$ **to** 255 **do**
      $pos(\mu_1) \leftarrow 1$
      $pr(\mu_1) \leftarrow E_{r-1}[\mu_1, 1] + \log(\lambda_{r,\mu_1,\mu_2})$
    for $i = 1$ **to** $\min(N, 256^{r-1})$ **do**
      $\mu_1 \leftarrow \mu$ which maximizes $pr(\mu)$
      $P_r[\mu_2, i] \leftarrow P_{r-1}[\mu_1, pos(\mu_1)] \,||\, \mu_2$
      $E_r[\mu_2, i] \leftarrow E_{r-1}[\mu_1, pos(\mu_1)] + \log(\lambda_{r,\mu_1,\mu_2})$
      $pos(\mu_1) \leftarrow pos(\mu_1) + 1$
      $pr(\mu_1) \leftarrow E_{r-1}[\mu_1, pos(\mu_1)] + \log(\lambda_{r,\mu_1,\mu_2})$
      if $pos(\mu_1) > \min(N, 256^{r-2})$ **then**
        $pr(\mu_1) \leftarrow -\infty$

return $P_N[m_L, :]$

algorithm) can be used, examples being [42, 36, 40, 28]. The simplest form of such an algorithm keeps track of the N best candidates ending in a particular value μ, and is shown in Algorithm 2. Similar to [2, 30] we assume the first byte m_1 and last byte m_L of the plaintext are known. During the last round of the outer for-loop, the loop over μ_2 has to be executed only for the value m_L. In Sect. 6 we use this algorithm to generate a list of cookies.

Algorithm 2 uses considerably more memory than Algorithm 1. This is because it has to store the N most likely candidates for each possible ending value μ. We remind the reader that our goal is not to present the most optimal algorithm. Instead, by showing how to model the problem as an HMM, we can rely on related work in this area for more efficient algorithms [42, 36, 40, 28]. Since an HMM can be modelled as a graph, all k-shortest path algorithms are also applicable [10]. Finally, we remark that even our simple variant sufficed to significantly improve plaintext recovery rates (see Sect. 6).

5 Attacking WPA-TKIP

We use our plaintext recovery techniques to decrypt a full packet. From this decrypted packet the MIC key can be derived, allowing an attacker to inject and decrypt packets. The attack takes only an hour to execute in practice.

5.1 Calculating Plaintext Likelihoods

We rely on the attack of Paterson et al. to compute plaintext likelihood estimates [31, 30]. They noticed that the first three bytes of the per-packet RC4 key are public. As explained in Sect. 2.2, the first three bytes are fully determined by the TKIP Sequence Counter (TSC). It was observed that this dependency causes strong TSC-dependent biases in the keystream [31, 15, 30], which can be used to improve the plaintext likelihood estimates. For each TSC value they calculated plaintext likelihoods based on empirical, per-TSC, keystream distributions. The resulting 256^2 likelihoods, for one plaintext byte, are combined by multiplying them over all TSC pairs. In a sense this is similar to combining multiple types of biases as done in Sect. 4.3, though here the different types of biases are known to be independent. We use the single-byte variant of the attack [30, §4.1] to obtain likelihoods $\lambda_{r,\mu}$ for every unknown byte at a given position r.

The downside of this attack is that it requires detailed per-TSC keystream statistics. Paterson at al. generated statistics for the first 512 bytes, which took 30 CPU years [30]. However, in our attack we only need these statistics for the first few keystream bytes. We used 2^{32} keys per TSC value to estimate the keystream distribution for the first 128 bytes. Using our distributed setup the generation of these statistics took 10 CPU years.

With our per-TSC keystream distributions we obtained similar results to that of Paterson et al. [31, 30]. By running simulations we confirmed that the odd byte positions [30], instead of the even ones [31], can be recovered with a higher probability than others. Similarly, the bytes at positions 49-51 and 63-67 are generally recovered with higher probability as well. Both observations will be used to optimize the attack in practice.

5.2 Injecting Identical Packets

We show how to fulfil the first requirement of a successful attack: the generation of identical packets. If the IP of the victim is know, and incoming connections towards it are not blocked, we can simply send identical packets to the victim. Otherwise we induce the victim into opening a TCP connection to an attacker-controlled server. This connection is then used to transmit identical packets to the victim. A straightforward way to accomplish this is by social engineering the victim into visiting a website hosted by the attacker. The browser will open a TCP connection with the server in order to load the website. However, we can also employ more sophisticated methods that have a broader target range. One

such method is abusing the inclusion of (insecure) third-party resources on popular websites [27]. For example, an attacker can register a mistyped domain, accidentally used in a resource address (e.g., an image URL) on a popular website. Whenever the victim visits this website and loads the resource, a TCP connection is made to the server of the attacker. In [27] these types of vulnerabilities were found to be present on several popular websites. Additionally, any type of web vulnerability that can be abused to make a victim execute JavaScript can be utilised. In this sense, our requirements are more relaxed than those of the recent attacks on SSL and TLS, which *require* the ability to run JavaScript code in the victim's browser [9, 1, 2]. Another method is to hijack an existing TCP connection of the victim, which under certain conditions is possible without a man-in-the-middle position [17]. We conclude that, while there is no universal method to accomplish this, we can assume control over a TCP connection with the victim. Using this connection we inject identical packets by repeatedly retransmitting identical TCP packets. Since retransmissions are valid TCP behaviour, this will work even if the victim is behind a firewall.

We now determine the optimal structure of the injected packet. A naive approach would be to use the shortest possible packet, meaning no TCP payload is included. Since the total size of the LLC/SNAP, IP, and TCP header is 48 bytes, the MIC and ICV would be located at position 49 up to and including 60 (see Fig. 2). At these locations 7 bytes are strongly biased. In contrast, if we use a TCP payload of 7 bytes, the MIC and ICV are located at position 56 up to and including 60. In this range 8 bytes are strongly biased, resulting in better plaintext likelihood estimates. Through simulations we confirmed that using a 7 byte payload increases the probability of successfully decrypting the MIC and ICV. In practice, adding 7 bytes of payload also makes the length of our injected packet unique. As a result we can easily identify and capture such packets. Given both these advantages, we use a TCP data packet containing 7 bytes of payload.

5.3 Decrypting a Complete Packet

Our goal is to decrypt the injected TCP packet, including its MIC and ICV fields. Note that all these TCP packets will be encrypted with a different RC4 key. For now we assume all fields in the IP and TCP packet are known, and we will later show why we can safely make this assumption. Hence, only the 8-byte MIC and 4-byte ICV of the packet remain unknown. We use the per-TSC keystream statistics to compute single-byte plaintext likelihoods for all 12 bytes. However, this alone would give a very low success probability of simultaneously (correctly) decrypting all bytes. We solve this by realising

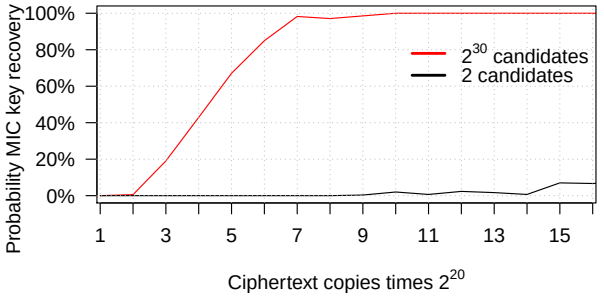

Figure 8: Success rate of obtaining the TKIP MIC key using nearly 2^{30} candidates, and using only the two best candidates. Results are based on 256 simulations each.

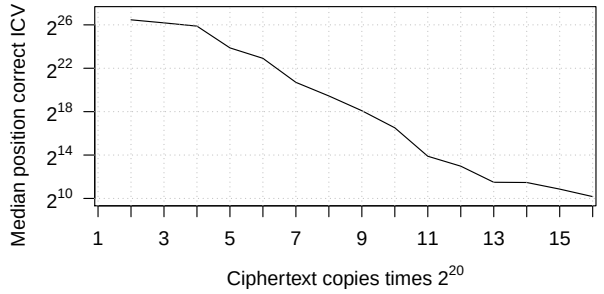

Figure 9: Median position of a candidate with a correct ICV with nearly 2^{30} candidates. Results are based on 256 simulations each.

that the TKIP ICV is a simple CRC checksum which we can easily verify ourselves. Hence we can detect bad candidates by inspecting their CRC checksum. We now generate a plaintext candidate list, and traverse it until we find a packet having a correct CRC. This drastically improves the probability of simultaneously decrypting all bytes. From the decrypted packet we can derive the TKIP MIC key [44], which can then be used to inject and decrypt arbitrary packets [48].

Figure 8 shows the success rate of finding a packet with a good ICV and deriving the correct MIC key. For comparison, it also includes the success rates had we only used the two most likely candidates. Figure 9 shows the median position of the first candidate with a correct ICV. We plot the median instead of average to lower influence of outliers, i.e., at times the correct candidate was unexpectedly far (or early) in the candidate list.

The question that remains how to determine the contents of the unknown fields in the IP and TCP packet. More precisely, the unknown fields are the internal IP and port of the client, and the IP time-to-live (TTL) field. One observation makes this clear: both the IP and TCP header contain checksums. Therefore, we can apply exactly the same technique (i.e., candidate generation and pruning) to derive the values of these fields with high

success rates. This can be done independently of each other, and independently of decrypting the MIC and ICV.

Another method to obtain the internal IP is to rely on HTML5 features. If the initial TCP connection is created by a browser, we can first send JavaScript code to obtain the internal IP of the victim using WebRTC [37]. We also noticed that our NAT gateway generally did not modify the source port used by the victim. Consequently we can simply read this value at the server. The TTL field can also be determined without relying on the IP checksum. Using a `traceroute` command we count the number of hops between the server and the client, allowing us to derive the TTL value at the victim.

5.4 Empirical Evaluation

To test the plaintext recovery phase of our attack we created a tool that parses a raw `pcap` file containing the captured Wi-Fi packets. It searches for the injected packets, extracts the ciphertext statistics, calculates plaintext likelihoods, and searches for a candidate with a correct ICV. From this candidate, i.e., decrypted injected packet, we derive the MIC key.

For the ciphertext generation phase we used an OpenVZ VPS as malicious server. The incoming TCP connection from the victim is handled using a custom tool written in Scapy. It relies on a patched version of Tcpreplay to rapidly inject the identical TCP packets. The victim machine is a Latitude E6500 and is connected to an Asus RT-N10 router running Tomato 1.28. The victim opens a TCP connection to the malicious server by visiting a website hosted on it. For the attacker we used a Compaq 8510p with an AWUS036nha to capture the wireless traffic. Under this setup we were able to generate roughly 2500 packets per second. This number was reached even when the victim was actively browsing YouTube videos. Thanks to the 7-byte payload, we uniquely detected the injected packet in all experiments without any false positives.

We ran several test where we generated and captured traffic for (slightly more) than one hour. This amounted to, on average, capturing $9.5 \cdot 2^{20}$ different encryptions of the packet being injected. Retransmissions were filtered based on the TSC of the packet. In nearly all cases we successfully decrypted the packet and derived the MIC key. Recall from Sect. 2.2 that this MIC key is valid as long as the victim does not renew its PTK, and that it can be used to inject and decrypt packets from the AP to the victim. For one capture our tool found a packet with a correct ICV, but this candidate did not correspond to the actual plaintext. While our current evaluation is limited in the number of captures performed, it shows the attack is practically feasible, with overall success probabilities appearing to agree with the simulated results of Fig. 8.

Listing 3: Manipulated HTTP request, with known plaintext surrounding the cookie at both sides.

```
1 GET / HTTP/1.1
2 Host: site.com
3 User-Agent: Mozilla/5.0 (X11; Linux i686; rv:32.0)
      Gecko/20100101 Firefox/32.0
4 Accept: text/html,application/xhtml+xml,application/
      xml;q=0.9,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Cookie: auth=XXXXXXXXXXXXXXXX; injected1=known1;
      injected2=knownplaintext2; ...
```

6 Decrypting HTTPS Cookies

We inject known data around a cookie, enabling use of the *ABSAB* biases. We then show that a HTTPS cookie can be brute-forced using only 75 hours of ciphertext.

6.1 Injecting Known Plaintext

We want to be able to predict the position of the targeted cookie in the encrypted HTTP requests, and surround it with known plaintext. To fix ideas, we do this for the secure `auth` cookie sent to `https://site.com`. Similar to previous attacks on SSL and TLS, we assume the attacker is able to execute JavaScript code in the victim's browser [9, 1, 2]. In our case, this means an active man-in-the-middle (MiTM) position is used, where plaintext HTTP channels can be manipulated. Our first realisation is that an attacker can predict the length and content of HTTP headers preceding the `Cookie` field. By monitoring plaintext HTTP requests, these headers can be sniffed. If the targeted `auth` cookie is the first value in the `Cookie` header, this implies we know its position in the HTTP request. Hence, our goal is to have a layout as shown in Listing 3. Here the targeted cookie is the first value in the `Cookie` header, preceded by known headers, and followed by attacker injected cookies.

To obtain the layout in Listing 3 we use our MiTM position to redirect the victim to `http://site.com`, i.e., to the target website over an insecure HTTP channel. If the target website uses HTTP Strict Transport Security (HSTS), but does not use the `includeSubDomains` attribute, this is still possible by redirecting the victim to a (fake) subdomain [6]. Since few websites use HSTS, and even fewer use it properly [47], this redirection will likely succeed. Against old browsers HSTS can even be bypassed completely [6, 5, 41]. Since secure cookies guarantee only confidentiality but not integrity, the insecure HTTP channel can be used to overwrite, remove, or inject secure cookies [3, 4.1.2.5]. This allows us to remove all cookies except the `auth` cookie, pushing it to the front of the list. After this we can inject cookies that

will be included after the `auth` cookie. An example of a HTTP(S) request manipulated in this manner is shown in Listing 3. Here the secure `auth` cookie is surrounded by known plaintext at both sides. This allows us to use Mantin's *ABSAB* bias when calculating plaintext likelihoods.

6.2 Brute-Forcing The Cookie

In contrast to passwords, many websites do not protect against brute-forcing cookies. One reason for this is that the password of an average user has a much lower entropy than a random cookie. Hence it makes sense to brute-force a password, but not a cookie: the chance of successfully brute-forcing a (properly generated) cookie is close to zero. However, if RC4 can be used to connect to the web server, our candidate generation algorithm voids this assumption. We can traverse the plaintext candidate list in an attempt to brute-force the cookie.

Since we are targeting a cookie, we can exclude certain plaintext values. As RFC 6265 states, a cookie value can consists of at most 90 unique characters [3, §4.1.1]. A similar though less general observation was already made by AlFardan et al. [2]. Our observation allows us to give a tighter bound on the required number of ciphertexts to decrypt a cookie, even in the general case. In practice, executing the attack with a reduced character set is done by modifying Algorithm 2 so the for-loops over μ_1 and μ_2 only loop over allowed characters.

Figure 10 shows the success rate of brute-forcing a 16-character cookie using at most 2^{23} attempts. For comparison, we also include the probability of decrypting the cookie if only the most likely plaintext was used. This also allows for an easier comparison with the work for AlFardan et al. [2]. Note that they only use the Fluhrer-McGrew biases, whereas we combine serveral *ABSAB* biases together with the Fluhrer-McGrew biases. We conclude that our brute-force approach, as well as the inclusion of the *ABSAB* biases, significantly improves success rates. Even when using only 2^{23} brute-force attempts, success rates of more than 94% are obtained once $9 \cdot 2^{27}$ encryptions of the cookie have been captured. We conjecture that generating more candidates will further increase success rates.

6.3 Empirical Evaluation

The main requirement of our attack is being able to collect sufficiently many encryptions of the cookie, i.e., having many ciphertexts. We fulfil this requirement by forcing the victim to generate a large number of HTTPS requests. As in previous attacks on TLS [9, 1, 2], we accomplish this by assuming the attacker is able to execute JavaScript in the browser of the victim. For example,

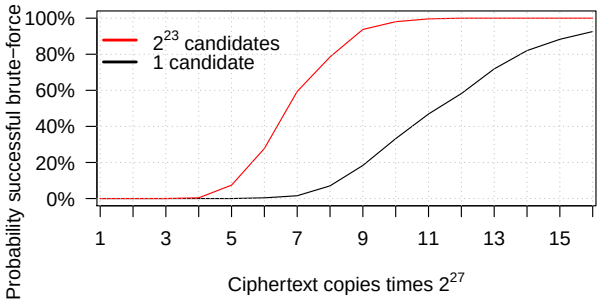

Figure 10: Success rate of brute-forcing a 16-character cookie using roughly 2^{23} candidates, and only the most likely candidate, dependent on the number of collected ciphertexts. Results based on 256 simulations each.

when performing a man-in-the-middle attack, we can inject JavaScript into any plaintext HTTP connection. We then use `XMLHttpRequest` objects to issue Cross-Origin Requests to the targeted website. The browser will automatically add the secure cookie to these (encrypted) requests. Due to the same-origin policy we cannot read the replies, but this poses no problem, we only require that the cookie is included in the request. The requests are sent inside HTML5 WebWorkers. Essentially this means our JavaScript code will run in the background of the browser, and any open page(s) stay responsive. We use GET requests, and carefully craft the values of our injected cookies so the targeted `auth` cookie is always at a fixed position in the keystream (modulo 256). Recall that this alignment is required to make optimal use of the Fluhrer-McGrew biases. An attacker can learn the required amount of padding by first letting the client make a request without padding. Since RC4 is a stream cipher, and no padding is added by the TLS protocol, an attacker can easily observe the length of this request. Based on this information it is trivial to derive the required amount of padding.

To test our attack in practice we implemented a tool in C which monitors network traffic and collects the necessary ciphertext statistics. This requires reassembling the TCP and TLS streams, and then detecting the 512-byte (encrypted) HTTP requests. Similar to optimizing the generation of datasets as in Sect. 3.2, we cache several requests before updating the counters. We also created a tool to brute-force the cookie based on the generated candidate list. It uses persistent connections and HTTP pipelining [11, §6.3.2]. That is, it uses one connection to send multiple requests without waiting for each response.

In our experiments the victim uses a 3.1 GHz Intel Core i5-2400 CPU with 8 GB RAM running Windows 7. Internet Explorer 11 is used as the browser. For the server a 3.4 GHz Intel Core i7-3770 CPU with 8 GB RAM is

used. We use nginx as the web server, and configured RC4-SHA1 with RSA as the only allowable cipher suite. This assures that RC4 is used in all tests. Both the server and client use an Intel 82579LM network card, with the link speed set to 100 Mbps. With an idle browser this setup resulted in an average of 4450 requests per second. When the victim was actively browsing YouTube videos this decreased to roughly 4100. To achieve such numbers, we found it's essential that the browser uses persistent connections to transmit the HTTP requests. Otherwise a new TCP and TLS handshake must be performed for every request, whose round-trip times would significantly slow down traffic generation. In practice this means the website must allow a `keep-alive` connection. While generating requests the browser remained responsive at all times. Finally, our custom tool was able to test more than 20000 cookies per second. To execute the attack with a success rate of 94% we need roughly $9 \cdot 2^{27}$ ciphertexts. With 4450 requests per seconds, this means we require 75 hours of data. Compared to the (more than) 2000 hours required by AlFardan et al. [2, §5.3.3] this is a significant improvement. We remark that, similar to the attack of AlFardan et al. [2], our attack also tolerates changes of the encryption keys. Hence, since cookies can have a long lifetime, the generation of this traffic can even be spread out over time. With 20000 brute-force attempts per second, all 2^{23} candidates for the cookie can be tested in less than 7 minutes.

We have executed the attack in practice, and successfully decrypted a 16-character cookie. In our instance, capturing traffic for 52 hours already proved to be sufficient. At this point we collected $6.2 \cdot 2^{27}$ ciphertexts. After processing the ciphertexts, the cookie was found at position 46229 in the candidate list. This serves as a good example that, if the attacker has some luck, less ciphertexts are needed than our $9 \cdot 2^{27}$ estimate. These results push the attack from being on the verge of practicality, to feasible, though admittedly somewhat time-consuming.

7 Related Work

Due to its popularity, RC4 has undergone wide cryptanalysis. Particularly well known are the key recovery attacks that broke WEP [12, 50, 45, 44, 43]. Several other key-related biases and improvements of the original WEP attack have also been studied [21, 39, 32, 22].

We refer to Sect. 2.1 for an overview of various biases discovered in the keystream [25, 23, 38, 2, 20, 33, 13, 24, 38, 15, 31, 30]. In addition to these, the long-term bias $\Pr[Z_r = Z_{r+1} \mid 2 \cdot Z_r = i_r] = 2^{-8}(1 + 2^{-15})$ was discovered by Basu et al. [4]. While this resembles our new short-term bias $\Pr[Z_r = Z_{r+1}]$, in their analysis they assume the internal state $\mathcal{S}$ is a random permutation, which is true only after a few rounds of the PRGA. Isobe et al. searched for dependencies between initial keystream bytes by empirically estimating $\Pr[Z_r = y \wedge Z_{r-a} = x]$ for $0 \leq x, y \leq 255$, $2 \leq r \leq 256$, and $1 \leq a \leq 8$ [20]. They did not discover any new biases using their approach. Mironov modelled RC4 as a Markov chain and recommended to skip the initial $12 \cdot 256$ keystream bytes [26]. Paterson et al. generated keystream statistics over consecutive keystream bytes when using the TKIP key structure [30]. However, they did not report which (new) biases were present. Through empirical analysis, we show that biases between consecutive bytes are present even when using RC4 with random 128 bit keys.

The first practical attack on WPA-TKIP was found by Beck and Tews [44] and was later improved by other researchers [46, 16, 48, 49]. Recently several works studied the per-packet key construction both analytically [15] and through simulations [2, 31, 30]. For our attack we replicated part of the results of Paterson et al. [31, 30], and are the first to demonstrate this type of attack in practice. In [2] AlFardan et al. ran experiments where the two most likely plaintext candidates were generated using single-byte likelihoods [2]. However, they did not present an algorithm to return arbitrarily many candidates, nor extended this to double-byte likelihoods.

The SSL and TLS protocols have undergone wide scrutiny [9, 41, 7, 1, 2, 6]. Our work is based on the attack of AlFardan et al., who estimated that $13 \cdot 2^{30}$ ciphertexts are needed to recover a 16-byte cookie with high success rates [2]. We reduce this number to $9 \cdot 2^{27}$ using several techniques, the most prominent being usage of likelihoods based on Mantin's *ABSAB* bias [24]. Isobe et al. used Mantin's *ABSAB* bias, in combination with previously decrypted bytes, to decrypt bytes after position 257 [20]. However, they used a counting technique instead of Bayesian likelihoods. In [29] a guess-and-determine algorithm combines *ABSAB* and Fluhrer-McGrew biases, requiring roughly 2^{34} ciphertexts to decrypt an individual byte with high success rates.

8 Conclusion

While previous attacks against RC4 in TLS and WPA-TKIP were on the verge of practicality, our work pushes them towards being practical and feasible. After capturing $9 \cdot 2^{27}$ encryptions of a cookie sent over HTTPS, we can brute-force it with high success rates in negligible time. By running JavaScript code in the browser of the victim, we were able to execute the attack in practice within merely 52 hours. Additionally, by abusing RC4 biases, we successfully attacked a WPA-TKIP network within an hour. We consider it surprising this is possible using only known biases, and expect these types of attacks to further improve in the future. Based on these results, we strongly urge people to stop using RC4.

9 Acknowledgements

We thank Kenny Paterson for providing valuable feedback during the preparation of the camera-ready paper, and Tom Van Goethem for helping with the JavaScript traffic generation code.

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] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[2] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the security of RC4 in TLS and WPA. In *USENIX Security Symposium*, 2013.

[3] A. Barth. HTTP state management mechanism. RFC 6265, 2011.

[4] R. Basu, S. Ganguly, S. Maitra, and G. Paul. A complete characterization of the evolution of RC4 pseudo random generation algorithm. *J. Mathematical Cryptology*, 2(3):257–289, 2008.

[5] D. Berbecaru and A. Lioy. On the robustness of applications based on the SSL and TLS security protocols. In *Public Key Infrastructure*, pages 248–264. Springer, 2007.

[6] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 98–113. IEEE, 2014.

[7] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a SSL/TLS channel. In *Advances in Cryptology (CRYPTO)*, 2003.

[8] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, 2008.

[9] T. Duong and J. Rizzo. Here come the xor ninjas. In *Ekoparty Security Conference*, 2011.

[10] D. Eppstein. k-best enumeration. *arXiv preprint arXiv:1412.5075*, 2014.

[11] R. Fielding and J. Reschke. Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. RFC 7230, 2014.

[12] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In *Selected areas in cryptography*. Springer, 2001.

[13] S. R. Fluhrer and D. A. McGrew. Statistical analysis of the alleged RC4 keystream generator. In *FSE*, 2000.

[14] C. Fuchs and R. Kenett. A test for detecting outlying cells in the multinomial distribution and two-way contingency tables. *J. Am. Stat. Assoc.*, 75:395–398, 1980.

[15] S. S. Gupta, S. Maitra, W. Meier, G. Paul, and S. Sarkar. Dependence in IV-related bytes of RC4 key enhances vulnerabilities in WPA. Cryptology ePrint Archive, Report 2013/476, 2013. `http://eprint.iacr.org/`.

[16] F. M. Halvorsen, O. Haugen, M. Eian, and S. F. Mjølsnes. An improved attack on TKIP. In *14th Nordic Conference on Secure IT Systems*, NordSec '09, 2009.

[17] B. Harris and R. Hunt. Review: TCP/IP security threats and attack methods. *Computer Communications*, 22(10):885–897, 1999.

[18] ICSI. The ICSI certificate notary. Retrieved 22 Feb. 2015, from `http://notary.icsi.berkeley.edu`.

[19] IEEE Std 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 2012.

[20] T. Isobe, T. Ohigashi, Y. Watanabe, and M. Morii. Full plaintext recovery attack on broadcast RC4. In *FSE*, 2013.

[21] A. Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3):269–286, 2008.

[22] S. Maitra and G. Paul. New form of permutation bias and secret key leakage in keystream bytes of RC4. In *Fast Software Encryption*, pages 253–269. Springer, 2008.

[23] S. Maitra, G. Paul, and S. S. Gupta. Attack on broadcast RC4 revisited. In *Fast Software Encryption*, 2011.

[24] I. Mantin. Predicting and distinguishing attacks on RC4 keystream generator. In *EUROCRYPT*, 2005.

[25] I. Mantin and A. Shamir. A practical attack on broadcast RC4. In *FSE*, 2001.

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

[27] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[28] D. Nilsson and J. Goldberger. Sequentially finding the n-best list in hidden Markov models. In *International Joint Conferences on Artificial Intelligence*, 2001.

[29] T. Ohigashi, T. Isobe, Y. Watanabe, and M. Morii. Full plaintext recovery attacks on RC4 using multiple biases. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 98(1):81–91, 2015.

[30] K. G. Paterson, B. Poettering, and J. C. Schuldt. Big bias hunting in amazonia: Large-scale computation and exploitation of RC4 biases. In *Advances in Cryptology — ASIACRYPT*, 2014.

[31] K. G. Paterson, J. C. N. Schuldt, and B. Poettering. Plaintext recovery attacks against WPA/TKIP. In *FSE*, 2014.

[32] G. Paul, S. Rathi, and S. Maitra. On non-negligible bias of the first output byte of RC4 towards the first three bytes of the secret key. *Designs, Codes and Cryptography*, 49(1-3):123–134, 2008.

[33] S. Paul and B. Preneel. A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher. In *FSE*, 2004.

[34] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2014.

[35] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 1989.

[36] M. Roder and R. Hamzaoui. Fast tree-trellis list Viterbi decoding. *Communications, IEEE Transactions on*, 54(3):453–461, 2006.

[37] D. Roesler. STUN IP address requests for WebRTC. Retrieved 17 June 2015, from `https://github.com/diafygi/webrtc-ips`.

[38] S. Sen Gupta, S. Maitra, G. Paul, and S. Sarkar. (Non-)random sequences from (non-)random permutations - analysis of RC4 stream cipher. *Journal of Cryptology*, 27(1):67–108, 2014.

[39] P. Sepehrdad, S. Vaudenay, and M. Vuagnoux. Discovery and exploitation of new biases in RC4. In *Selected Areas in Cryptography*, pages 74–91. Springer, 2011.

[40] N. Seshadri and C.-E. W. Sundberg. List Viterbi decoding algorithms with applications. *IEEE Transactions on Communications*, 42(234):313–323, 1994.

[41] B. Smyth and A. Pironti. Truncating TLS connections to violate beliefs in web applications. In *WOOT'13: 7th USENIX Workshop on Offensive Technologies*, 2013.

[42] F. K. Soong and E.-F. Huang. A tree-trellis based fast search for finding the n-best sentence hypotheses in continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 705–708. IEEE, 1991.

[43] A. Stubblefield, J. Ioannidis, and A. D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *ACM Trans. Inf. Syst. Secur.*, 7(2), 2004.

[44] E. Tews and M. Beck. Practical attacks against WEP and WPA. In *Proceedings of the second ACM conference on Wireless network security*, WiSec '09, 2009.

[45] E. Tews, R.-P. Weinmann, and A. Pyshkin. Breaking 104 bit WEP in less than 60 seconds. In *Information Security Applications*, pages 188–202. Springer, 2007.

[46] Y. Todo, Y. Ozawa, T. Ohigashi, and M. Morii. Falsification attacks against WPA-TKIP in a realistic environment. *IEICE Transactions*, 95-D(2), 2012.

[47] T. Van Goethem, P. Chen, N. Nikiforakis, L. Desmet, and W. Joosen. Large-scale security analysis of the web: Challenges and findings. In *TRUST*, 2014.

[48] M. Vanhoef and F. Piessens. Practical verification of WPA-TKIP vulnerabilities. In *ASIACCS*, 2013.

[49] M. Vanhoef and F. Piessens. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC*, 2014.

[50] S. Vaudenay and M. Vuagnoux. Passive–only key recovery attacks on RC4. In *Selected Areas in Cryptography*, pages 344–359. Springer, 2007.