On the Security of RC4 in TLS

Nadhem AlFardan, *Royal Holloway, University of London;*
Daniel J. Bernstein, *University of Illinois at Chicago and Technische Universiteit Eindhoven;*
Kenneth G. Paterson, Bertram Poettering, and Jacob C.N. Schuldt,
Royal Holloway, University of London

On the Security of RC4 in TLS[1]

Nadhem J. AlFardan
Information Security Group,
Royal Holloway, University of London

Daniel J. Bernstein
University of Illinois at Chicago and
Technische Universiteit Eindhoven

Kenneth G. Paterson
Information Security Group,
Royal Holloway, University of London

Bertram Poettering
Information Security Group,
Royal Holloway, University of London

Jacob C. N. Schuldt
Information Security Group,
Royal Holloway, University of London

Abstract

The Transport Layer Security (TLS) protocol aims to provide confidentiality and integrity of data in transit across untrusted networks. TLS has become the *de facto* protocol standard for secured Internet and mobile applications. TLS supports several symmetric encryption options, including a scheme based on the RC4 stream cipher. In this paper, we present ciphertext-only plaintext recovery attacks against TLS when RC4 is selected for encryption. Our attacks build on recent advances in the statistical analysis of RC4, and on new findings announced in this paper. Our results are supported by an experimental evaluation of the feasibility of the attacks. We also discuss countermeasures.

1 Introduction

TLS is arguably the most widely used secure communications protocol on the Internet today. Starting life as SSL, the protocol was adopted by the IETF and specified as an RFC standard under the name of TLS 1.0 [7]. It has since evolved through TLS 1.1 [8] to the current version TLS 1.2 [9]. Various other RFCs define additional TLS cryptographic algorithms and extensions. TLS is now used for securing a wide variety of application-level traffic: It serves, for example, as the basis of the HTTPS protocol for encrypted web browsing, it is used in conjunction with IMAP or SMTP to cryptographically protect email traffic, and it is a popular tool to secure communication with embedded systems, mobile devices, and in payment systems.

Technically speaking, TLS sessions consist of two consecutive phases: the execution of the TLS Handshake Protocol which typically deploys asymmetric techniques to establish a secure session key, followed by the execution of the TLS Record Protocol which uses symmetric key cryptography (block ciphers, the RC4 stream cipher, MAC algorithms) in combination with the established session key and sequence numbers to build a se-

cure channel for transporting application-layer data. In the Record Protocol, there are mainly three encryption options:

- HMAC followed by CBC-mode encryption using a block cipher,

- HMAC followed by encryption using the RC4 stream cipher, or

- authenticated encryption using GCM or CCM mode of operation of a block cipher.

The third of these three options is only available with TLS 1.2 [21, 18], which is yet to see widespread adoption.[2] The first option has seen significant cryptanalysis (padding oracle attacks [6], BEAST [10], Lucky 13 [3]). While countermeasures to the attacks on CBC-mode in TLS exist, many commentators now recommend, and many servers now offer, RC4-based encryption options ahead of CBC-mode.[3] Indeed, the ICSI Certificate Notary[4] recently performed an analysis of 16 billion TLS connections and found that around 50% of the traffic was protected using RC4 ciphersuites [5].

This makes it timely to examine the security of RC4 in TLS. While the RC4 algorithm is known to have a variety of cryptographic weaknesses (see [23] for an excellent survey), it has not been previously explored how these weaknesses can be exploited in the context of TLS. Here we show that new and recently discovered biases in the RC4 keystream do create serious vulnerabilities in TLS when using RC4 as its encryption algorithm.

While the main focus of this paper lies on the security of RC4 in TLS, our attacks (or variants thereof) might also be applicable to other protocols where RC4 is meant to ensure data confidentiality. Indeed, the WPA protocol used for encrypting wireless network traffic also utilizes the RC4 stream cipher in a way that allows (partial) plaintext recovery in specific settings — using basically the same attack strategies as in the TLS case.

We hope that this work will help spur the adoption of TLS 1.2 and its authenticated encryption algorithms, as well as the transition from WPA to (the hopefully more secure) WPA2.

1.1 Overview of Results

We present two plaintext recovery attacks on RC4 that are exploitable in specific but realistic circumstances when this cipher is used for encryption in TLS. Both attacks require a fixed plaintext to be RC4-encrypted and transmitted many times in succession (in the same, or in multiple independent RC4 keystreams). Interesting candidates for such plaintexts include passwords and, in the setting of secure web browsing, HTTP cookies.

A statistical analysis of ciphertexts forms the core of our attacks. We stress that the attacks are ciphertext-only: no sophisticated timing measurement is needed on the part of the adversary, the attacker does not need to be located close to the server, and no packet injection capability is required (all premises for Lucky 13). Instead, it suffices for the adversary to record encrypted traffic for later offline analysis. Provoking the required repeated encryption and transmission of the target plaintext, however, might require more explicit action: e.g., resetting TCP connections or guiding the victim to a website with specially prepared JavaScript (see examples below).

Since both our attacks require large amounts of ciphertext, their practical relevance could be questioned. However, they do show that the strength of RC4 in TLS is much lower than the employed 128-bit key would suggest. We freely admit that our attacks are not particularly deep, nor sophisticated: they only require an understanding of how TLS uses RC4, solid statistics on the biases in RC4 keystreams, and some experience of how modern browsers handle cookies. We consider it both surprising and alarming that such simple attacks are possible for such an important and heavily-studied protocol as TLS. We further discuss the implications of our attack in Section 6 and in the full version of this paper [4].

1.1.1 Our single-byte bias attack

Our first attack targets the initial 256 bytes of RC4 ciphertext. It is fixed-plaintext and multi-session, meaning that it requires a fixed sequence of plaintext bytes to be independently encrypted under a large number of (random) keys. This setting corresponds to what is called a "broadcast attack" in [17, 15, 23]. As we argue below, such attacks are a realistic attack vector in TLS. Observe that, in TLS, the first 36 bytes of the RC4 keystream are used to encrypt a TLS Handshake Finished message. This message is not fixed across TLS sessions. As a consequence, our methods can be applied only to recover up

to 220 bytes of the TLS application plaintext.

Our attack exploits statistical biases occurring in the first 256 bytes of RC4 keystream. Such biases, i.e., deviations from uniform in the distributions of the keystream bytes at certain positions, have been reported and theoretically analyzed by [17], [15], and [23]. The corresponding authors also propose algorithms to exploit such biases for plaintext recovery. In this paper, we discuss shortcomings of their algorithms, empirically obtain a complete view of all single-byte biases occurring in the first 256 keystream positions, and propose a generalized algorithm that fully exploits all these biases for advanced plaintext recovery. As a side result of our research, in Section 3.1 we report on significant biases in the RC4 keystream that seemingly follow specific patterns and that have not been identified or analysed previously.

For concreteness, we describe how our single-byte bias attack could be applied to recover cookies in HTTPS traffic. Crucial here is to find an automated mechanism for efficiently generating a large number of encryptions of the target cookie. In line with the scenario employed by the BEAST and Lucky 13 attacks against CBC-mode encryption in TLS [3, 10], a candidate mechanism is for JavaScript malware downloaded from an attacker-controlled website and running in the victim's browser to repeatedly send HTTPS requests to a remote server. The corresponding cookies are automatically included in each of these requests in a predictable location, and can thus be targeted in our attack. If client and server are configured to use TLS session resumption, the renewal of RC4 keys could be arranged to happen with particularly high frequency — as required for our attack to be successful.[5] Alternatively, the attacker can cause the TLS session to be terminated after the target encrypted cookie is sent; the browser will automatically establish a new TLS session when the next HTTPS request is sent.

As a second example, consider the case where IMAP passwords[6] are attacked. In a setup where an email client regularly connects to an IMAP server for (password-authenticated) mail retrieval, let the adversary reset the TCP connection between client and server immediately after the encrypted password is transmitted. In some client configurations this might trigger an automatic resumption of the session, including a retransmission of the (encrypted) password. If this is the case, the adversary is in the position to harvest a large set of independently encrypted copies of the password —one per reset— precisely fulfilling the precondition of our attack.

Our single-byte bias attack is on the verge of practicality. In our experiments, the first 40 bytes of TLS application data after the Finished message were recovered with a success rate of over 50% per byte, using 2^{26} sessions. With 2^{32} sessions, the per-byte success rate is more than 96% for the first 220 bytes (and is 100%

for all but 12 of these bytes). If, for example, a target plaintext byte is known to be a character from a set of cardinality 16 (e.g., in a 4-bits-per-byte-encoded HTTP cookie), our algorithm recovers the first 112 bytes of plaintext with a success rate of more than 50% per byte, using 2^{26} sessions. For further details, see Section 5.

1.1.2 Our double-byte bias attack

As we have seen, our single-byte bias attack on RC4 is quite effective in recovering 'early' plaintext bytes in the fixed-plaintext multi-session setting. It has, however, a couple of limitations when it comes to attacking practical systems that employ TLS. Focussing on the recovery of cookies in HTTPS-secured web sessions, we note that modern web browsers typically send a large number of HTTP headers before any cookies (these headers carry information about the particular client or server software, accepted MIME types, compression options, etc.). In practice, cookie data appears only at positions that come after the attackable initial 220 bytes of the ciphertext[7]. Independently of this issue, in the attack scenarios proposed above, a large number of HTTPS sessions would have to be established and torn down again, inducing non-negligible computing and bandwidth overheads via the TLS Handshake. Lastly, it has been proposed to routinely drop the first few hundred keystream bytes of RC4 before starting encryption in order to avoid the relatively strong early keystream biases [19] — if this were to be implemented in TLS, our single-byte bias attack would effectively be defeated.

Complementary to our single-byte bias attack, we present a second fixed-plaintext ciphertext-only attack on RC4. It exploits biases that appear in the entire keystream (and not just in the first 256 positions) and does not assume, but tolerates, frequent changes of the encryption key. Our second attack hence covers some scenarios where our single-byte bias attack does not seem to be applicable; it would, for example, be able to recover cookies from (long-persisting) HTTPS sessions. It would also be applicable if the initial keystream bytes were to be discarded.

In contrast to our first attack, our second attack exploits certain biases in consecutive pairs of bytes in the RC4 keystream that were first reported by Fluhrer and McGrew [12]. We empirically evaluate the probability of occurrence for each possible pair of bytes beginning at each position (modulo 256), obtaining a complete view of the distributions of pairs of bytes in positions $(i, i+1)$ (modulo 256). Our analysis strongly suggests that there are no further biases in consecutive positions of the same strength as the Fluhrer-McGrew biases. We use the obtained results in a specially designed attack algorithm to recover repeatedly encrypted plaintexts.

Our double-byte bias attack is again close to being practical. In our experiments, we focus on our attack's ability to correctly recover 16 consecutive bytes of plaintext, roughly equating to an HTTP cookie. With $13 \cdot 2^{30}$ encryptions of the plaintext, we achieve a success rate of 100% in recovering all 16 bytes. We obtain better success rates for restricted plaintexts, as in the single-byte case. For further details, see Section 5.

1.2 Related Work

In independent and concurrent work, Isobe *et al.* [13] have considered the security of RC4 against broadcast attacks. They present attacks based on both single-byte and multi-byte biases. They identify three biases in the first output bytes Z_r of RC4 that we also identify (specifically, the biases towards $Z_3 = 0x83$, $Z_r = r$, and $Z_r = -r$ when r is a multiple of 16) as well as a new conditional bias $Z_1 = 0 | Z_2 = 0$.

The single-byte bias attack in [13] only considers the *strongest* bias at each position, whereas our single-byte bias attack simultaneously exploits *all* biases in each keystream position. Specifically, we use Bayes's law to compute the *a posteriori* plaintext distribution from the *a priori* plaintext distribution and the precomputed distributions of the Z_r. This explains why our single-byte attack out-performs that of [13]. For example, we achieve reliable plaintext recovery in the first 256 positions with 2^{32} ciphertexts, while Isobe *et al.* [13] require 2^{34} ciphertexts. We also achieve uniformly higher success rates for lower numbers of sessions. Previous authors exploring broadcast attacks on RC4 also only used single biases, leading to attacks that simply do not work [15, 23] or which have inferior performance to ours [22].

The multi-byte bias attack in [13] exploits the positive bias towards the pattern $AB\mathcal{S}AB$ that was identified by Mantin [16]. Here A and B are keystream bytes and $\mathcal{S}$ is a short string consisting of any keystream bytes (possibly of length 0). The attack in [13] assumes that 3-out-of-4 bytes in particular positions are known and uses the Mantin bias to recover the fourth. A limited experimental evaluation of the attack is reported in [13]: the attack is applied only to recovery of plaintext bytes 258-261, assuming all previous plaintext bytes have been successfully recovered, with success rates of 1 (for each of the 4 targeted bytes) using 2^{34} ciphertexts. As explained in [13], this multi-byte attack would fail if the initial bytes of RC4 output were to be discarded. By contrast, our double-byte bias attack, which exploits the Fluhrer-McGrew biases, recovers more bytes with comparable success rate using slightly fewer ciphertexts and is resilient to initial byte discarding. It is an interesting open problem to determine whether the Mantin $AB\mathcal{S}AB$ bias can be combined with the Fluhrer-McGrew biases to

gain enhanced attack performance.

A further point of comparison between our work and that of [13] concerns practical implementation. We have extensively explored the applicability of our attacks to RC4 as used in TLS, while [13] makes only brief mention of TLS in its concluding section and gives no mechanisms for generating the large numbers of ciphertexts needed for the attacks.

Finally, the authors of [13] claim in their abstract that their methods *"can recover the first 2^{50} bytes $\approx 1000\ T$ bytes of the plaintext, with probability close to 1, from only 2^{34} ciphertexts"*. We point out that their methods would only recover 2^{16} *distinct* bytes of output, rather than the advertised 2^{50} bytes, since their attacks require the *same* plaintext to be encrypted 2^{34} times. Furthermore, their multi-byte bias attack is not resilient to errors occurring in the recovery of early plaintext bytes (whereas ours is), so this claim would only be true if their multi-byte bias attack does not fail at any stage, and this is as yet untested.

1.3 Paper Organisation

Section 2 provides further background on the RC4 stream cipher and the TLS Record Protocol. Section 3 summarises weaknesses in RC4 that we exploit in our attacks. Section 4 describes our two plaintext recovery attacks on RC4. We evaluate the attacks in Section 5, with our main focus there being on TLS. Finally, Section 6 discusses countermeasures to our attacks, and concludes with a recap of the main issues raised by our work.

2 Further Background

2.1 The RC4 Stream Cipher

The stream cipher RC4, originally designed by Ron Rivest, became public in 1994 and found application in a wide variety of cryptosystems; well-known examples include SSL/TLS, WEP [1], WPA [2], and some Kerberos-related encryption modes [14]. RC4 has a remarkably short description and is extremely fast when implemented in software. However, these advantages come at the price of lowered security: several weaknesses have been identified in RC4 [12, 11, 17, 16, 15, 23, 25, 24, 26], some of them being confirmed and exploited in the current paper.

Technically, RC4 consists of two algorithms: a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA), which are specified in Figure 1. The KSA takes as input a key K, typically a byte-array of length between 5 and 32 (i.e., 40 to 256 bits), and produces the initial internal state $st_0 = (i, j, \mathcal{S})$, where $\mathcal{S}$ is the canonical representation of a permutation on the set

Algorithm 1: RC4 key scheduling (KSA)

> **input** : key K of l bytes
> **output**: internal state st_0
> **begin**
> > **for** $i = 0$ **to** 255 **do**
> > > $\mathcal{S}[i] \leftarrow i$
> >
> > $j \leftarrow 0$
> > **for** $i = 0$ **to** 255 **do**
> > > $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$
> > > swap$(\mathcal{S}[i], \mathcal{S}[j])$
> >
> > $i, j \leftarrow 0$
> > $st_0 \leftarrow (i, j, \mathcal{S})$
> > **return** st_0

Algorithm 2: RC4 keystream generator (PRGA)

> **input** : internal state st_r
> **output**: keystream byte Z_{r+1}
> > internal state st_{r+1}
> **begin**
> > parse $(i, j, \mathcal{S}) \leftarrow st_r$
> > $i \leftarrow i + 1$
> > $j \leftarrow j + \mathcal{S}[i]$
> > swap$(\mathcal{S}[i], \mathcal{S}[j])$
> > $Z_{r+1} \leftarrow \mathcal{S}[\mathcal{S}[i] + \mathcal{S}[j]]$
> > $st_{r+1} \leftarrow (i, j, \mathcal{S})$
> > **return** (Z_{r+1}, st_{r+1})

Figure 1: Algorithms implementing the RC4 stream cipher. All additions are performed modulo 256.

$[0, 255]$ as an array of bytes, and i, j are indices into this array. The PRGA will, given an internal state st_r, output 'the next' keystream byte Z_{r+1}, together with the updated internal state st_{r+1}. Particularly interesting to note is the fact that updated index j is computed in dependence on current i, j, and $\mathcal{S}$, while i is just a counter (modulo 256).

2.2 The TLS Record Protocol

We describe in detail the cryptographic operation of the TLS Record Protocol in the case that RC4 is selected as the encryption method.

Data to be protected by TLS is received from the application and may be fragmented and compressed before further processing. An individual record R (viewed as a sequence of bytes) is then processed as follows. The sender maintains an 8-byte sequence number SQN which is incremented for each record sent, and forms a 5-byte field HDR consisting of a 2-byte version field, a 1-byte type field, and a 2-byte length field. It then calculates an HMAC over the string HDR||SQN||R; let T denote the resulting tag.

For RC4 encryption, record and tag are concatenated to create the plaintext $P = R||T$. This plaintext is then xored in a byte-by-byte fashion using the RC4 keystream, i.e., the ciphertext bytes are computed as

$$C_r = P_r \oplus Z_r \qquad \text{for } r = 1, 2, 3, \ldots ,$$

where P_r are the individual bytes of P, and Z_r are the RC4 keystream bytes. The data transmitted over the wire then has the form

$$HDR||C ,$$

where C is the concatenation of the bytes C_r.

The RC4 algorithm itself is initialised at the start of each TLS connection, using a 128 bit encryption key K. This key K is computed with a hash-function-based key

derivation function from the TLS master secret that is established during the TLS Handshake Protocol. In more detail, the key K may be established either via a full TLS Handshake or via TLS session resumption. In a full TLS Handshake, a total of 4 communication round-trips are needed, and usually some public key cryptographic operations are required of both client and server. A full TLS Handshake run establishes a new TLS *session* and a new TLS master secret from which all other keys, including RC4 key K, are derived. TLS session resumption involves a lightweight version of the TLS Handshake Protocol being run to establish a new *connection* within an existing session: essentially, an exchange of nonces takes place, followed by an exchange of `Finished` messages; no public key cryptographic operations are involved. The keys for the new connection, including K, are derived from the existing master secret and the new nonces. Given the design of the key derivation process, it is reasonable to model K as being uniformly random in the different sessions/connections.

The initialisation of RC4 in TLS is the standard one for this algorithm. Notably, none of the initial keystream bytes is discarded when RC4 is used in TLS, despite these bytes having known weaknesses. Note also that the first record sent under the protection of RC4 for each session or connection will be a `Finished` message, typically of length 36 bytes, consisting of a Handshake Protocol header, a PRF output, and a MAC on that output. This is typically 36 bytes in size. This record will not be targeted in our attacks, since it is not constant across multiple sessions.

The decryption process reverses this sequence of steps, but its details are not germane to our attacks. For TLS, any error arising during decryption should be treated as fatal, meaning an (encrypted) error message is sent to the sender and the session terminated with all keys and other cryptographic material being disposed of. This gives an attacker a convenient method to cause a session to be terminated and force new encryption and MAC keys to be set up. Another method is to somehow induce the client or server to initiate session resumption.

3 Biases in the RC4 Keystream

In this section, we summarise known biases in the RC4 keystream, and report new biases that we have observed experimentally.

3.1 Single-byte Biases

The first significant bias in the RC4 keystream was observed by Mantin and Shamir in [17]. Their main result can be stated as:

Result 1. *[17, Thm 1] The probability that Z_2, the second byte of keystream output by RC4, is equal to* `0x00` *is approximately* $1/128$ *(where the probability is taken over the random choice of the key).*

Since this result concerns only the second byte of the keystream, and this byte is always used to encrypt a `Finished` message in TLS, we are unable to exploit it in our attacks. More recently, the following result was obtained by Sen Gupta *et al.* in [23] as a refinement of an earlier result of Maitra *et al.* [15]:

Result 2. *[23, Thm 14 and Cor 3] For $3 \le r \le 255$, the probability that Z_r, the r-th byte of keystream output by RC4, is equal to* `0x00` *is*

$$\Pr(Z_r = \texttt{0x00}) = \frac{1}{256} + \frac{c_r}{256^2} \ ,$$

where the probability is taken over the random choice of the key, $c_3 = 0.351089$, and $c_4, c_5 \ldots, c_{255}$ is a decreasing sequence with terms that are bounded as follows:

$$0.242811 \le c_r \le 1.337057.$$

In other words, bytes 3 to 255 of the keystream have a bias towards `0x00` of approximately $1/2^{16}$. This result was experimentally verified in [23] and found to be highly accurate (see Figure 11 of that paper). The biases here are substantially smaller than those observed in Result 1.

Additionally, Sen Gupta *et al.* [23] have identified a key-length-dependent bias in RC4 keystreams. Specifically, [23, Theorem 5] shows that when the key-length is ℓ bytes, then byte Z_ℓ is biased towards value $256 - \ell$, with the bias always being greater than $1/2^{16}$. For RC4 in TLS, we have $\ell = 16$.

Experimentally, we have observed additional biases in the RC4 keystream that do not yet have a theoretical explanation. As an example, Figure 2 shows the empirical distribution for the RC4 keystream bytes Z_{16}, Z_{32} and Z_{50}, calculated over 2^{44} independent, random 128-bit keys. For Z_{16}, we have 3 main biases: the bias towards `0x00`, the very dominant key-length-dependent bias towards `0xF0` (decimal 240) from [23], and a new bias towards `0x10` (decimal 16). For Z_{32}, we also have 3 main biases: the bias towards `0x00`, a large, new bias towards `0xE0` (decimal 224), and a new bias towards `0x20` (decimal 32). For Z_{50}, there are significant biases towards byte values `0x00` and `0x32` (decimal 50), as well as an upward trend in probability as the byte value increases.

Individual inspection of ciphertext distributions at all positions $1 \le r \le 256$ reveals two new significant biases that occur with specific regularities: a bias towards value r for all r, and a bias towards value $256 - r$ at positions r that are *multiples* of (key-length) 16; note that
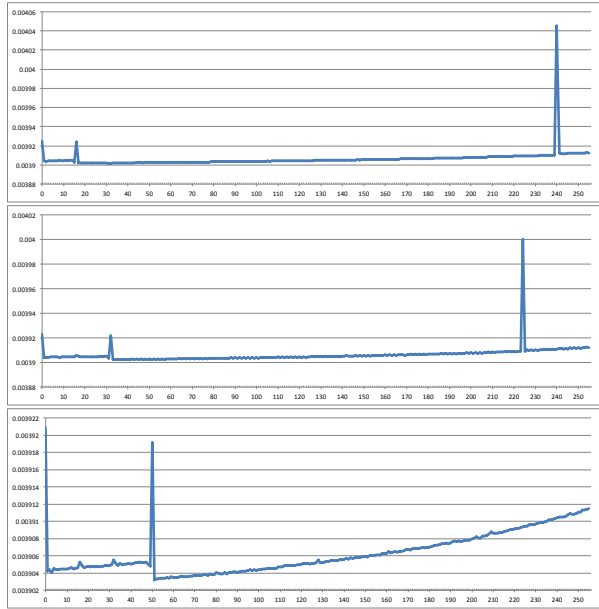
Figure 2: Measured distributions of RC4 keystream bytes Z_{16} (top), Z_{32} (middle), and Z_{50} (bottom).

the latter finding both confirms and extends the results from [23]. Both of these new biases were also observed by Isobe *et al.* [13], with a theoretical explanation being given for the bias towards r. Figure 3 shows the estimated strength of these biases in comparison with the strength of the bias towards 0x00 for the keystream bytes $Z_1, \ldots, Z_{256}$. The estimates are based on the empirical distribution of the RC4 keystream bytes, calculated over 2^{44} random 128-bit RC4 keys. We note that the key-length dependent bias dominates the other two biases until position Z_{112}, and that the bias of Z_r towards r dominates the bias towards 0x00 observed by [15] between positions Z_5 and Z_{31}, except for byte Z_{16} where the bias towards 0x00 is slightly stronger.

Furthermore, for the first keystream byte Z_1, we have observed a bias *away* from value 0x81 (decimal 129) in the addition to the known bias away from value 0x00. This additional bias is not consistent with the recent results of Sen Gupta *et al.* [23] who provide a theoretical treatment of the distribution of Z_1. The disparity likely arises because Sen Gupta *et al.* work with 256-byte keys, while our work is exclusively concerned with 128-bit (16-byte) keys as used in TLS; in other words, our observed bias in $Z_1 = 0x81$ seems to be key-length-dependent. Finally, our computations have revealed a number of other, smaller biases in the initial bytes of the RC4 keystream.
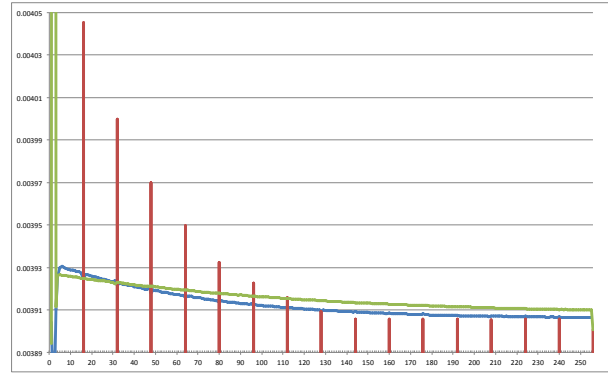

Figure 3: Measured strength of the bias towards 0x00 (green), the bias towards value r in Z_r (blue), and the key-length dependent bias towards byte value $256 - r$ (red) for keystream bytes $Z_1, \ldots, Z_{256}$, based on keystreams generated by 2^{44} independent random keys. Note that the large peak for the 0x00 bias in Z_2 extends beyond the bounds of the graph and is not fully shown for illustrative purposes.

3.2 Multi-byte Biases

Besides the single-byte biases highlighted above, several multi-byte biases have been identified in the RC4 keystream. In contrast to the single-byte biases, most of the identified multi-byte biases are "long term" biases which appear periodically at regular intervals in the keystream.

The most extensive set of multi-byte biases was identified by Fluhrer and McGrew [12] who analyzed the distribution of pairs of byte values for consecutive keystream positions (Z_r, Z_{r+1}), $r \geq 1$. More precisely, they estimated the distribution of consecutive keystream bytes for scaled-down[8] versions of RC4 by assuming an idealized internal state of RC4 in which the permutation S and the internal variable j are random (see Figure 1), and then extrapolated the results to standard RC4.

The reported biases for standard RC4 are listed in Table 1. Note that all biases are dependent on the internal variable i which is incremented (modulo 256) for each keystream byte generated. It should also be noted that, due to the assumption that S and j are random, the biases cannot be expected to hold for the initial keystream bytes. However, this idealization becomes a close approximation to the internal state of RC4 after a few invocations of the RC4 keystream generator, [12].

We experimentally verified the Fluhrer-McGrew biases by analysing the output of 2^{10} RC4 instances using 128-bit keys and generating 2^{40} keystream bytes each. For each keystream, the initial 1024 bytes were dropped. Based on this data, we found the biases from [12] to be accurate, also for 128-bit keys. This is in-line with the

experiments and observations reported in [12]. Furthermore, we did not identify any additional significant long term biases for consecutive keystream bytes which are repeated with a periodicity that is a proper divisor of 256. Hence, for the purpose of implementing the attack presented in Section 4.2, we assume that the biases identified in [12] are the only existing long term biases for consecutive keystream bytes, and that all other pairs of byte-values are uniformly distributed.

Byte pair	Condition on i	Probability
$(0,0)$	$i = 1$	$2^{-16}(1+2^{-9})$
$(0,0)$	$i \neq 1, 255$	$2^{-16}(1+2^{-8})$
$(0,1)$	$i \neq 0, 1$	$2^{-16}(1+2^{-8})$
$(i+1, 255)$	$i \neq 254$	$2^{-16}(1+2^{-8})$
$(255, i+1)$	$i \neq 1, 254$	$2^{-16}(1+2^{-8})$
$(255, i+2)$	$i \neq 0, 253, 254, 255$	$2^{-16}(1+2^{-8})$
$(255, 0)$	$i = 254$	$2^{-16}(1+2^{-8})$
$(255, 1)$	$i = 255$	$2^{-16}(1+2^{-8})$
$(255, 2)$	$i = 0, 1$	$2^{-16}(1+2^{-8})$
$(129, 129)$	$i = 2$	$2^{-16}(1+2^{-8})$
$(255, 255)$	$i \neq 254$	$2^{-16}(1-2^{-8})$
$(0, i+1)$	$i \neq 0, 255$	$2^{-16}(1-2^{-8})$

Table 1: Fluhrer-McGrew biases for consecutive pairs of byte values. In the table, i is the internal variable of the RC4 keystream generation algorithm (see Section 2.1).

Independently of [12], Mantin [16] identified a positive bias towards the pattern $AB\mathcal{S}AB$, where A and B represent byte values and $\mathcal{S}$ is a short string of bytes (possibly of length 0). The shorter the string $\mathcal{S}$ is, the more significant is the bias. Additionally, Sen Gupta *et al.* [23] identified a bias towards the byte values $(0,0)$ for keystream positions (Z_r, Z_{r+2}), separated by any single keystream byte for $r \geq 1$. However, we do not make use of these biases in the attacks presented in this paper.

4 Plaintext Recovery Attacks

For the purpose of exposition, we first explain how the broadcast attack by Maitra *et al.* [15] and Sen Gupta *et al.* [23] is meant to work. Suppose byte Z_r of the RC4 keystream has a dominant bias towards value 0x00. As RC4 encryption is defined as $C_r = P_r \oplus Z_r$, the corresponding ciphertext byte C_r has a bias towards plaintext byte P_r. Thus, obtaining sufficiently many ciphertext samples C_r for a fixed plaintext P_r allows inference of P_r by a majority vote: P_r is equal to the value of C_r that occurs most often. This is the core idea of Algorithm 3 that we reproduce from [15, 23]. Let S denote the number of ciphertexts available to the attacker and, for all $1 \leq j \leq S$, let $C_{j,r}$ denote the r-th byte of ciphertext C_j. For a fixed position r, Algorithm 3 runs through all j, and in each iteration increments one out of 256 counters,

---

Algorithm 3: Basic plaintext recovery attack

input : S independent encryptions $(C_j)_{1 \leq j \leq S}$ of fixed plaintext P, position r
output: estimate P_r^* for plaintext byte P_r
begin
  $N_{\texttt{0x00}} \leftarrow 0, \ldots, N_{\texttt{0xFF}} \leftarrow 0$
  for $j = 1$ **to** S **do**
    $N_{C_{j,r}} \leftarrow N_{C_{j,r}} + 1$
  $P_r^* \leftarrow \arg\max_{\mu \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\}} N_\mu$

---

namely the one that corresponds to value $C_{j,r}$. After processing all ciphertexts, the character corresponding to the largest counter in the obtained histogram is the output of the algorithm.

The algorithm is tailor-made for plaintext recovery in the case described by Result 2: it assumes that the largest bias in the RC4 keystream is towards 0x00. However, it is highly likely to fail to reliably suggest the correct plaintext byte P_r if the RC4 keystream has, in position r, additional biases of approximately the same size (or larger) as the bias towards 0x00. Such additional biases would simply be misinterpreted as the bias towards 0x00 and hence falsify the result. As we observed in Section 3.1 (and Figure 3), several other quite strong biases in the RC4 keystream do indeed exist. This clearly invalidates Algorithm 3 for practical use.

4.1 Our Single-byte Bias Attack

We propose a plaintext-recovery algorithm that takes into account *all* possible single-byte RC4 biases at the same time, along with their strengths. The idea is to first obtain a detailed picture of the distributions of RC4 keystream bytes Z_r, for all positions r, by gathering statistics from keystreams generated using a large number of independent keys (2^{44} in our case). That is, for all r, we (empirically) estimate

$$p_{r,k} := \Pr(Z_r = k), \quad k = \texttt{0x00}, \ldots, \texttt{0xFF} ,$$

where the probability is taken over the random choice of the RC4 encryption key (i.e., 128 bit keys in the TLS case). Using these biases $p_{r,k}$, in a second step, plaintext can be recovered with optimal accuracy using a maximum-likelihood approach, as follows.

Suppose we have S ciphertexts $C_1, \ldots, C_S$ available for our attack. For any fixed position r and any candidate plaintext byte μ for that position, vector $(N_{\texttt{0x00}}^{(\mu)}, \ldots, N_{\texttt{0xFF}}^{(\mu)})$ with

$$N_k^{(\mu)} = |\{j \mid C_{j,r} = k \oplus \mu\}_{1 \leq j \leq S}| \quad (\texttt{0x00} \leq k \leq \texttt{0xFF})$$

represents the distribution on Z_r required to obtain the observed ciphertexts $\{C_{j,r}\}_{1 \leq j \leq S}$ by encrypting μ. We

Algorithm 4: Single-byte bias attack

input : S independent encryptions $\{C_j\}_{1 \leq j \leq S}$ of fixed plaintext P, position r, keystream distribution $(p_{r,k})_{\texttt{0x00} \leq k \leq \texttt{0xFF}}$ at position r

output: estimate P_r^* for plaintext byte P_r

begin

    $N_{\texttt{0x00}} \leftarrow 0, \ldots, N_{\texttt{0xFF}} \leftarrow 0$

    for $j = 1$ **to** S **do**

        $N_{C_{j,r}} \leftarrow N_{C_{j,r}} + 1$

    for $\mu = \texttt{0x00}$ **to** $\texttt{0xFF}$ **do**

        for $k = \texttt{0x00}$ **to** $\texttt{0xFF}$ **do**

            $N_k^{(\mu)} \leftarrow N_{k \oplus \mu}$

        $\lambda_\mu \leftarrow \sum_{k=\texttt{0x00}}^{\texttt{0xFF}} N_k^{(\mu)} \log p_{r,k}$

    $P_r^* \leftarrow \arg\max_{\mu \in \{\texttt{0x00},\ldots,\texttt{0xFF}\}} \lambda_\mu$

    return P_r^*

compare these *induced* distributions (one for each possible μ) with the accurate distribution $p_{r,\texttt{0x00}}, \ldots, p_{r,\texttt{0xFF}}$ and interpret a close match as an indication for the corresponding plaintext candidate μ being the correct one, i.e., $P_r = \mu$. More formally, we observe that the probability λ_μ that plaintext byte μ is encrypted to ciphertext bytes $\{C_{j,r}\}_{1 \leq j \leq S}$ follows a multinomial distribution and can be precisely calculated as

$$\lambda_\mu = \frac{S!}{N_{\texttt{0x00}}^{(\mu)}! \cdots N_{\texttt{0xFF}}^{(\mu)}!} \prod_{k \in \{\texttt{0x00},\ldots,\texttt{0xFF}\}} p_{r,k}^{N_k^{(\mu)}} \quad . \quad (1)$$

By computing λ_μ for all $\texttt{0x00} \leq \mu \leq \texttt{0xFF}$ and identifying μ such that λ_μ is largest, we determine the (optimal) maximum-likelihood plaintext byte value. Algorithm 4 specifies the details of the described single-byte bias attack, including the optimizations discussed next.

Observe that, for each fixed position r and set of ciphertexts $\{C_{j,r}\}_{1 \leq j \leq S}$, values $N_k^{(\mu)}$ can be computed from values $N_k^{(\mu')}$ by equation $N_k^{(\mu)} = N_{k \oplus \mu' \oplus \mu}^{(\mu')}$, for all k. In other words, vectors $(N_{\texttt{0x00}}^{(\mu)}, \ldots, N_{\texttt{0xFF}}^{(\mu)})$ and $(N_{\texttt{0x00}}^{(\mu')}, \ldots, N_{\texttt{0xFF}}^{(\mu')})$ are permutations of each other; by consequence, term $S!/(N_{\texttt{0x00}}^{(\mu)}! \cdots N_{\texttt{0xFF}}^{(\mu)}!)$ in equation (1) can safely be ignored when determining the largest λ_μ. Furthermore, computing and comparing $\log(\lambda_\mu)$ instead of λ_μ makes the computation slightly more efficient.

4.2 Our Double-byte Bias Attack

As we have seen, Algorithm 4 allows the recovery of the initial 256 bytes of plaintext when multiple encryptions under different keys are observed by the attacker. In the following, we describe an algorithm which allows the recovery of plaintext bytes at *any* position in the plaintext.

Furthermore, the algorithm does not require the plaintext to be encrypted under many different keys but works equally well for plaintexts repeatedly encrypted under a *single* key.

Our algorithm is based on biases in the distribution of consecutive bytes (Z_r, Z_{r+1}) of the RC4 keystream that occur as long term biases, i.e., that appear periodically at regular intervals in the keystream. As described in Section 3, we empirically measured the biases which are repeated with a period of 256 bytes. However, in 2^{50} experimentally generated keystream bytes we observed no significant new biases besides those already identified by Fluhrer and McGrew [12]; for the purpose of constructing our algorithm, we hence use the biases described in Table 1 and assume that all other consecutive byte pairs are equally likely to appear in the keystream. In other words, we assume that we have accurate estimates p_{r,k_1,k_2} such that

$$p_{r,k_1,k_2} = \Pr[(Z_r, Z_{r+1}) = (k_1, k_2)]$$

for $1 \leq r \leq 256$ and $\texttt{0x00} \leq k_1, k_2 \leq \texttt{0xFF}$, where the probability is taken over all possible configurations of the internal state $\mathcal{S}$ and the index j of the RC4 keystream generation algorithm.[9] Note that, since these probabilities express biases that are repeated with a period of 256 bytes, we have $p_{r,k_1,k_2} = p_{(r \bmod 256),k_1,k_2}$ for all r, k_1, k_2.

Let L be an integer multiple of 256. In the following description of our plaintext recovery algorithm, we assume that a fixed L-byte plaintext $P = P_1 || \cdots || P_L$ is encrypted repeatedly under a single key, i.e., we consider a ciphertext C obtained by encrypting $P || \cdots || P$. (In fact, it is sufficient for our attack that the target plaintext bytes form a subsequence of consecutive bytes that are constant across blocks of L bytes.) Let C_j denote the substring of C corresponding to the encryption of the j-th copy of P, and let $C_{j,r}$ denote the r-th byte of C_j (i.e., $C_{j,r}$ corresponds to byte $(j-1) \cdot L + r$ of C).

Given this setting, it seems reasonable to take an approach towards plaintext recovery similar to that of Algorithm 4: for each position r, the most likely plaintext pair (μ_r, μ_{r+1}) could be computed from the ciphertext bytes $\{(C_{j,r}, C_{j,r+1})\}_{1 \leq j \leq S}$ and the probability estimates $\{p_{r,k_1,k_2}\}_{\texttt{0x00} \leq k_1,k_2 \leq \texttt{0xFF}}$. In other words, a plaintext candidate would be obtained by splitting ciphertexts C into byte pairs and individually computing the most likely corresponding plaintext pairs.

However, by considering overlapping byte pairs, it is possible to construct a more accurate estimate of the likelihood of a plaintext candidate being correct than by just considering the likelihood of individual byte-pairs. More specifically, for any plaintext candidate $P' = \mu_1 || \cdots || \mu_L$ we compute an estimated likelihood $\lambda_{P'} = \lambda_{\mu_1 || \cdots || \mu_L}$ for

P' being correct via the recursion

$$\lambda_{\mu_1||\cdots||\mu_{\ell-1}||\mu_\ell} = \delta_{\mu_\ell|\mu_{\ell-1}} \cdot \lambda_{\mu_1||\cdots||\mu_{\ell-1}} \qquad (\ell \leq L), \quad (2)$$

where $\delta_{\mu_\ell|\mu_{\ell-1}}$ denotes the probability that $P_\ell = \mu_\ell$ assuming $P_{\ell-1} = \mu_{\ell-1}$, and $\lambda_{\mu_1||\cdots||\mu_{\ell-1}}$ is the estimated likelihood of $\mu_1||\cdots||\mu_{\ell-1}$ being the correct $(\ell-1)$-length prefix of P. We show below how values $\delta_{\mu_\ell|\mu_{\ell-1}}$ can be computed given the ciphertext bytes $\{(C_{j,\ell-1}, C_{j,\ell})\}_{1\leq j\leq S}$ and the probability estimates $\{p_{\ell-1,k_1,k_2}\}_{\texttt{0x00}\leq k_1,k_2\leq\texttt{0xFF}}$. Note that, by rewriting equation (2) and assuming that $\lambda_{\mu_1} = \Pr[P_1 = \mu_1]$ is accurately known, we obtain likelihood estimate $\lambda_{P'} = \Pr[P_1 = \mu_1]\prod_{\ell=2}^{L} \delta_{\mu_\ell|\mu_{\ell-1}}$.

Our algorithm computes the plaintext candidate $P^* = \mu_1||\cdots||\mu_L$ which maximizes the estimated likelihood λ_{P^*}. This is done by exploiting the following easy-to-see optimality-preserving property: for all prefixes $\mu_1||\cdots||\mu_\ell$ of P^*, $\ell \leq L$, we have that $\lambda_{\mu_1||\cdots||\mu_{\ell-1}}$ is the largest likelihood among all $(\ell-1)$-length plaintext candidates with $\mu_{\ell-1}$ as the last byte.

The basic idea of our algorithm is to iteratively construct P^* by considering the prefixes of P^* with increasing length. As just argued, these correspond to the (partial) plaintext candidates with the highest likelihood and a specific choice of the last byte value. However, when computing a candidate for a length $\ell \leq L$, it is not known in advance what the specific value of the last byte μ_ℓ should be. Our algorithm hence computes the most likely partial plaintext candidates for *all* possible values of μ_ℓ. More specifically, for each $(\ell-1)$-length partial candidate $\mu_1||\cdots||\mu_{\ell-1}$ and any value μ_ℓ, we compute the likelihood of the ℓ-length plaintext candidate $\mu_1||\cdots||\mu_{\ell-1}||\mu_\ell$ via equation (2) as $\lambda_{\mu_1||\cdots||\mu_\ell} = \delta_{\mu_\ell|\mu_{\ell-1}} \cdot \lambda_{\mu_1||\cdots||\mu_{\ell-1}}$. Due to the optimality-preserving property, the string $\mu_1||\cdots||\mu_\ell$ with the highest likelihood will correspond to the most likely plaintext candidates of length ℓ with the last byte μ_ℓ. This guarantees that the ℓ-length prefix of (optimal) P^* will be among the computed candidates and, furthermore, when the length of P^* is reached, that P^* itself will be obtained.

To initialize the above process, the algorithm assumes that the first plaintext byte μ_1 of P is known with certainty, i.e., $\lambda_{\mu_1} = 1$ (this can, for example, be assumed if the attack is used to recover HTTP cookies from an encrypted HTTP(S) header). Likewise, the algorithm assumes that the last byte μ_L of P is known, i.e., $\lambda_{\mu_L} = 1$ (also this is the case when recovering HTTP cookies). This leads to a single μ_L being used in the last iteration of the above process which will then return the most likely plaintext candidate P^*. (See Remark 1 for how the algorithm can be modified to work without these assumptions.)

It remains to show the details of how $\delta_{\mu_{i+1}|\mu_i}$ can be computed. This is done similarly to the maximum-likelihood computation of the probability estimate used in Algorithm 4. More precisely, each combination of index i, pair (μ_i,μ_{i+1}), and ciphertext bytes $\{(C_{j,i},C_{j,i+1})\}_{1\leq j\leq S}$ induces a distribution on the keystream bytes $\{(Z_{(j-1)L+i},Z_{(j-1)L+i+1})\}_{1\leq j\leq S}$. The latter can be represented as a vector $(N_{i,\texttt{0x00},\texttt{0x00}},\ldots,N_{i,\texttt{0xFF},\texttt{0xFF}})$, where

$$N_{i,k_1,k_2} = |\{j \,|\, (C_{j,i}, C_{j,i+1}) = (k_1\oplus\mu_i, k_2\oplus\mu_{i+1})\}_{1\leq j\leq S}| \,.$$

As in Section 4.1, we see that this vector follows a multinomial distribution, and that the probability that $(N_{i,\texttt{0x00},\texttt{0x00}},\ldots,N_{i,\texttt{0xFF},\texttt{0xFF}})$ will arise (i.e., the probability that (μ_i,μ_{i+1}) corresponds to the i-th and the $(i+1)$-th plaintext bytes) is given by

$$\Pr[P_i = \mu_i \wedge P_{i+1} = \mu_{i+1} \,|\, C] = \qquad (3)$$

$$\frac{S!}{N_{i,\texttt{0x00},\texttt{0x00}}!\cdots N_{i,\texttt{0xFF},\texttt{0xFF}}!} \prod_{k_1,k_2\in\{\texttt{0x00},\ldots,\texttt{0xFF}\}} p_{i,k_1,k_2}^{N_{i,k_1,k_2}} \,.$$

We can now compute $\delta_{\mu_{i+1}|\mu_i}$ as

$$\begin{aligned}\delta_{\mu_{i+1}|\mu_i} &= \Pr[P_{i+1} = \mu_{i+1} \,|\, P_i = \mu_i \wedge C] \\ &= \frac{\Pr[P_i = \mu_i \wedge P_{i+1} = \mu_{i+1} \,|\, C]}{\Pr[P_i = \mu_i \,|\, C]} \,. \quad (4)\end{aligned}$$

We assume that no significant single-byte biases are present in the keystream, i.e., that $\Pr[P_i = \mu_i \,|\, C]$ is uniform over the possible plaintext values μ_i. Under this condition, since the term will stay invariant for all plaintext candidates, we can ignore the contribution of factor $1/\Pr[P_i = \mu_i \,|\, C]$ in (4), when comparing probability estimates. This is likewise the case for the terms $S!/(N_{i,\texttt{0x00},\texttt{0x00}},\ldots,N_{i,\texttt{0xFF},\texttt{0xFF}})$ in (3), due to similar observations as made for Algorithm 4.

We combine the results of the discussion from the preceeding paragraphs, including the proposed optimizations, to obtain our double-byte bias attack in Algorithm 5.

Remark 1. The above assumption, that the first and last byte of the plaintext P is known, can easily be avoided. Specifically, if the first byte is unknown, Algorithm 5 can be initialized by computing, for each possible value μ_2, the most likely pairs (μ_1,μ_2). This can be done based on the ciphertext bytes $\{(C_{j,1},C_{j,2})\}_{1\leq j\leq S}$ and the probability estimates $\{p_{1,k_1,k_2}\}_{\texttt{0x00}\leq k_1,k_2\leq\texttt{0xFF}}$. Likewise, if the last byte is unknown, the algorithm will identify P^* as the plaintext candidate with the highest likelihood estimate among the computed plaintext candidates of length L. Note, however, that knowing the first and last plaintext byte will lead to a more accurate likelihood estimate and will thereby increase the success rate of the algorithm.

Algorithm 5: Double-byte bias attack

---

input : C – encryption of S copies of fixed plaintext P
        ($C_{j,r}$ denotes the r-th byte of the substring of C encrypting the j-th copy of P)
        L – length of P in bytes (must be a multiple of 256)
        μ_1 and μ_L – the first and last byte of P
        $\{p_{r,k_1,k_2}\}_{1 \leq r \leq L-1,\, \texttt{0x00} \leq k_1, k_2 \leq \texttt{0xFF}}$ – keystream distribution
output: estimate P^* for plaintext P
notation: let $\max_2(Q)$ denote $(P,\lambda) \in Q$ such that $\lambda \geq \lambda' \, \forall (P', \lambda') \in Q$
begin

    $N_{(r,k_1,k_2)} \leftarrow 0$ for all $1 \leq r < L$, $\texttt{0x00} \leq k_1, k_2 \leq \texttt{0xFF}$
    for $j = 1$ **to** S **do**
        for $r = 1$ **to** $L - 1$ **do**
            $N_{(r,C_{j,r},C_{j,r+1})} \leftarrow N_{(r,C_{j,r},C_{j,r+1})} + 1$

    $Q \leftarrow \{(\mu_1, 0)\}$
    for $r = 1$ **to** $L - 2$ **do**
        $Q_{ext} \leftarrow \{\}$ `// List of plaintext candidates of length` $r+1$
        for $\mu_{r+1} = \texttt{0x00}$ **to** $\texttt{0xFF}$ **do**
            $Q_{\mu_{r+1}} \leftarrow \{\}$ `// List of plaintext candidates ending with` μ_{r+1}
            for each $(P', \lambda_{P'}) \in Q$ **do**
                $P' \rightarrow \mu_1 || \cdots || \mu_r$
                $\lambda_{P' || \mu_{r+1}} \leftarrow \lambda_{P'} + \sum_{k_1 = \texttt{0x00}}^{\texttt{0xFF}} \sum_{k_2 = \texttt{0x00}}^{\texttt{0xFF}} N_{(r, k_1 \oplus \mu_r, k_2 \oplus \mu_{r+1})} \cdot \log p_{(r,k_1,k_2)}$
                $Q_{\mu_{r+1}} \leftarrow Q_{\mu_{r+1}} \cup \{(P' || \mu_{r+1}, \lambda_{P' || \mu_{r+1}})\}$
            $Q_{ext} \leftarrow Q_{ext} \cup \{\max_2(Q_{\mu_{r+1}})\}$
        $Q \leftarrow Q_{ext}$

    $Q_{\mu_L} \leftarrow \{\}$ `// List of plaintext candidates ending with` μ_L
    for each $(P', \lambda_{P'}) \in Q$ **do**
        $P' \rightarrow \mu_1 || \cdots || \mu_{L-1}$
        $\lambda_{P' || \mu_L} \leftarrow \lambda_{P'} + \sum_{k_1 = \texttt{0x00}}^{\texttt{0xFF}} \sum_{k_2 = \texttt{0x00}}^{\texttt{0xFF}} N_{(r, k_1 \oplus \mu_{L-1}, k_2 \oplus \mu_L)} \cdot \log p_{(r,k_1,k_2)}$
        $Q_{\mu_L} \leftarrow Q_{\mu_L} \cup \{(P' || \mu_L, \lambda_{P' || \mu_L})\}$
    $(P^*, \lambda_{P^*}) \leftarrow \max_2(Q_{\mu_L})$
    return P^*

---

5   Experimental Results

Through simulation, we measured the performance of the single-byte and double-byte bias attacks. We furthermore validated our algorithms in real attack scenarios.

5.1   Simulation of Single-byte Bias Attack

We simulated the first plaintext recovery attack described in Section 4. We used RC4 keystreams for 2^{44} random keys to estimate the per-output-byte probabilities $\{p_{r,k}\}_{1 \leq r \leq 256,\, \texttt{0x00} \leq k \leq \texttt{0xFF}}$. We then ran the attack in Algorithm 4 256 times for each of $S = 2^{24}, 2^{25}, \ldots, 2^{32}$ sessions to estimate the attack's success rate. The results for $S = 2^{24}, 2^{26}, \ldots, 2^{30}$ are shown in Figures 4–7. In each figure, we show the success rate in recovering the correct plaintext byte versus the position r of the byte in the output stream (but recall that, in practice, the first 36 bytes

are not interesting as they contain the `Finished` message). Some notable features of these figures are:

- Even with as few as 2^{24} sessions, some positions of the plaintext are correctly recovered with high probability. The ones with highest probability seem to arise because of the key-length-dependent biases that we observed in positions that are multiples of 16. These large biases make it easier to recover the correct plaintext bytes when compared to other ciphertext positions.

- With $S = 2^{26}$ sessions, the first 46 plaintext bytes are recovered with rate at least 50% per byte.

- With $S = 2^{32}$ sessions (not shown here; see [4]), all of the first 256 bytes of output are recovered with rate close to 100%: the rate is at least 96% in all positions, and is 100% for all but 12 positions.
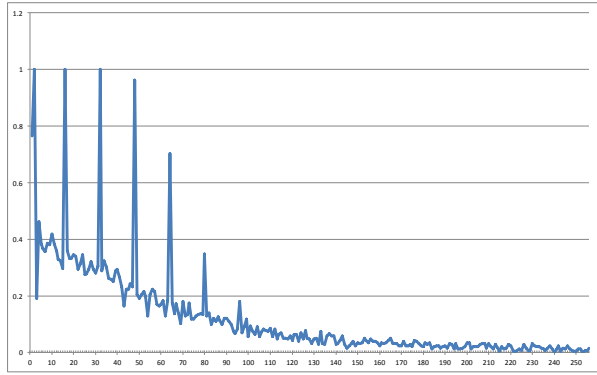
Figure 4: Recovery rate of the single-byte bias attack for $S = 2^{24}$ sessions for first 256 bytes of plaintext (based on 256 experiments).
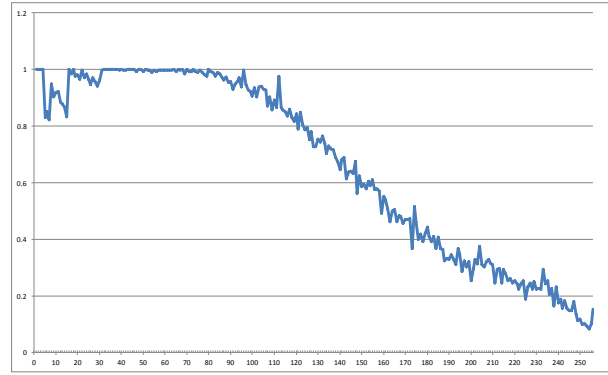

Figure 6: Recovery rate of the single-byte bias attack for $S = 2^{28}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).
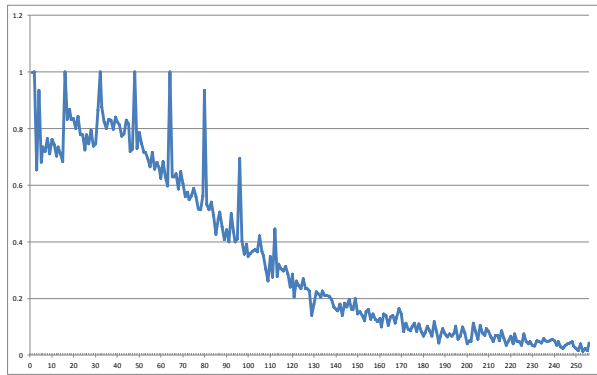

Figure 5: Recovery rate of the single-byte bias attack for $S = 2^{26}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).
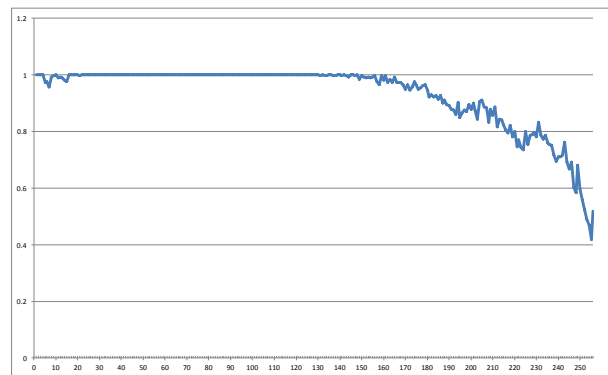

Figure 7: Recovery rate of the single-byte bias attack for $S = 2^{30}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).

- The rate at which bytes are correctly recovered increases steadily as the number of sessions S is increased, with all but the last few bytes being reliably recovered already for 2^{31} trials.

Secondly, we executed the recovery attack in a setting where plaintexts are encoded with a 4-bits-per-byte encoding scheme using the characters '0' to '9' and 'a' to 'f'. Such restricted plaintext character sets are routinely used in different applications [4]; for instance, in the popular PHP server-side scripting language, the encoding of HTTP cookies can be limited to a representation with 4 bits per character [20]. We reused the probability estimates $\{p_{r,k}\}_{1 \leq r \leq 256, \mathtt{0x00} \leq k \leq \mathtt{0xFF}}$ for the RC4 keystream bytes generated for the simulation above, and ran a modified version of Algorithm 4 which takes into account the restricted plaintext space. The modified algorithm was run 256 times for each of $S = 2^{24}, 2^{25}, \ldots, 2^{32}$ sessions. The results for $S = 2^{24}$, $S = 2^{26}$ and $S = 2^{28}$ are shown in Figures 8–10. For comparison, the figures include the success rate of the original attack

for an unrestricted plaintext space. We note:

- With $S = 2^{26}$ sessions, the first 112 plaintext bytes are recovered with rate at least 50% per byte. This represents a marked improvement over the case of an unrestricted plaintext space, where only the first 46 bytes were recovered with rate at least 50% per byte.

- With $S = 2^{24}, \ldots, 2^{28}$ sessions, the recovery attack for the restricted plaintext space has a better success rate than the recovery attack for the unrestricted plaintext space with twice the number of sessions (i.e. $S = 2^{25}, \ldots, 2^{29}$) for almost all positions.

5.2 Simulation of Double-byte Bias Attack

We simulated the second plaintext recovery attack based on Algorithm 5. In the simulation, we encrypted $S = 1 \cdot 2^{30}, \ldots, 13 \cdot 2^{30}$ copies of the same 256-byte plaintext
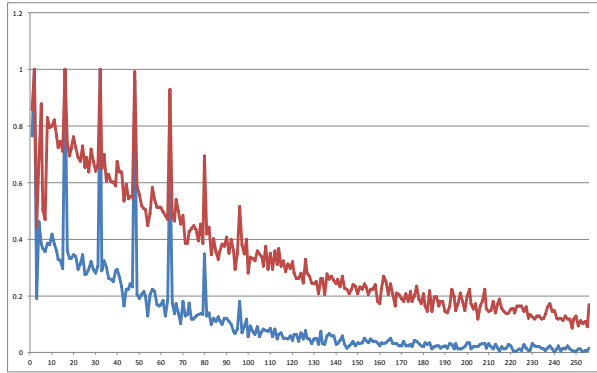
Figure 8: Recovery rates for the restricted plaintext space (red) and the original single-byte bias attack (blue) for $S = 2^{24}$ sessions (based on 256 experiments).
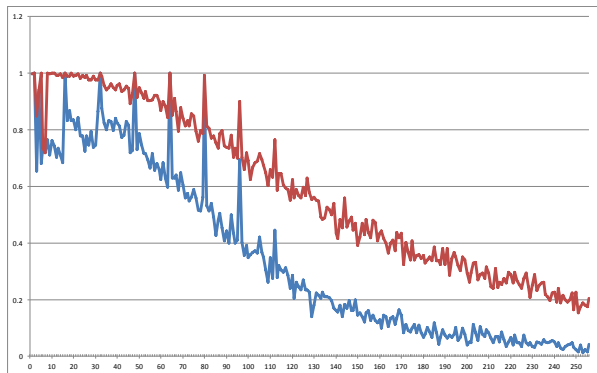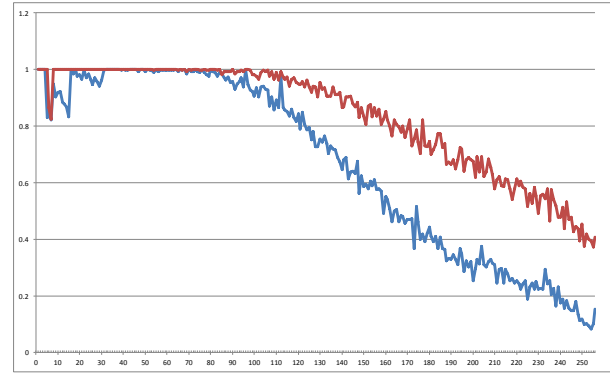

Figure 10: Recovery rates for the restricted plaintext space (red) and the original single-byte bias attack (blue) for $S = 2^{28}$ sessions (based on 256 experiments).

Figure 9: Recovery rates for the restricted plaintext space (red) and the original single-byte bias attack (blue) for $S = 2^{26}$ sessions (based on 256 experiments).

and attempted to recover 16 bytes located at a fixed position in the plaintext. More precisely, we simulated an attack in which we assume the first byte of the plaintext is known, the following 16 bytes are the unknown bytes targeted by the attack, and the byte immediately following these is known. The remaining bytes are assumed not to be of interest in the attack. This attack scenario is very similar to the case in which an adversary attempts to recover a cookie value from an HTTP request. Depending on the number of plaintext copies, we used between one and five 128-bit RC4 keys for the encryption[10]. As highlighted in Section 4.2, we used the biases described by Fluhrer-McGrew [12] to compute the probability estimates $\{p_{r,k_1,k_2}\}_{1 \leq r \leq 255, \mathtt{0x00} \leq k_1, k_2 \leq \mathtt{0xFF}}$ required by Algorithm 5.

The attack was run 128 times for each of $S = 1 \cdot 2^{30}, \ldots, 13 \cdot 2^{30}$ encrypted copies of the plaintext to estimate the success rate of the attack. The results are shown in Figure 11: the dashed line shows the average fraction of successfully recovered plaintext bytes versus the num-

ber of encrypted plaintexts, whereas the solid line shows the success rate of recovering the full 16-byte plaintext versus the number of encrypted plaintexts. We note:

- With $S = 6 \cdot 2^{30}$ encrypted copies of the plaintext, more than 50% of the plaintext is correctly recovered on average. Furthermore, in 19% of the 128 trials, the full 16-byte plaintext was recovered.

- With $S = 8 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is correctly recovered in significantly more than 50% of the 128 trials (more precisely, the full plaintext was recovered in 72% of the trials).

- With $S = 13 \cdot 2^{30}$ the full plaintext was recovered in all trials.

- The rate at which the full plaintext is correctly recovered increases fairly rapidly after $S = 5 \cdot 2^{30}$ copies of the plaintext are encrypted, and with $S = 11 \cdot 2^{30}$, the full plaintext is correctly recovered in nearly all trials (99%).

In addition, similar to Section 5.1, we simulated the attack for plaintexts encoded with a 6-bits-per-byte (base64) and a 4-bits-per-byte encoding scheme. Specifically, we firstly ran a modified version of Algorithm 5 which takes into account the restricted plaintext space by only considering candidate plaintext bytes which correspond to byte-values used in a base64 encoding. Furthermore, we used a plaintext where the 16 bytes targeted by the attack consisted of bytes with a byte-value corresponding to the character 'b', which is a valid base64 encoded message. As in the attack above for a non-restricted plaintext space, the probability estimates $\{p_{r,k_1,k_2}\}_{1 \leq r \leq 255, \mathtt{0x00} \leq k_1, k_2 \leq \mathtt{0xFF}}$ were based on the biases from [12]. The attack was run 128 times for each of
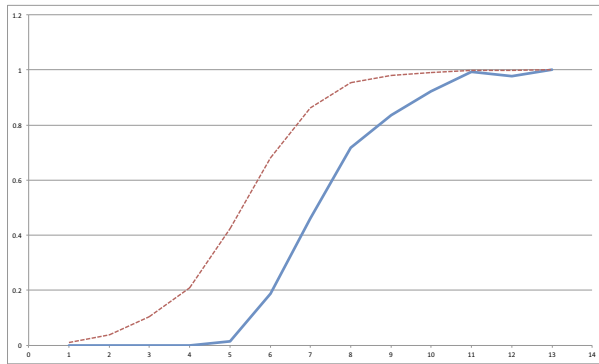
Figure 11: Average fraction of successfully recovered plaintext bytes (dashed line), and success rate for recovering the full 16-byte plaintext (solid line) of the double-byte bias attack based on 128 experiments. The unit of the x-axis is 2^{30} encrypted copies of the plaintext.
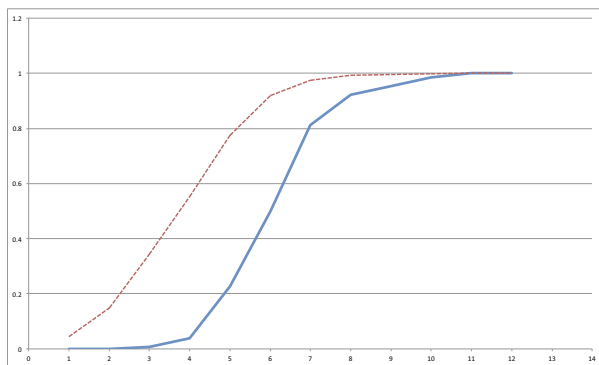

Figure 12: Average fraction of successfully recovered plaintext bytes (dashed line), and success rate for recovering the full 16-byte plaintext (solid line) of the double-byte bias attack for base64 encoded plaintexts (based on 128 experiments). The unit of the x-axis is 2^{30} encrypted copies of the plaintext.

$S = 1 \cdot 2^{30}, \ldots, 12 \cdot 2^{30}$ encrypted copies of the plaintext, and the results are shown in Figure 12. We note:

- With $S = 4 \cdot 2^{30}$ encrypted copies of the plaintext, more than 50% of the plaintext is correctly recovered on average. Furthermore, in 4% of the 128 trials, the full 16-byte plaintext is recovered.

- With $S = 6 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is correctly recovered in 50% of the 128 trials.

- With $S = 10 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is correctly recovered in nearly all trials (98%).

Regarding the 4-bit-per-byte encoding scheme, we again assumed a plaintext character set consisting of
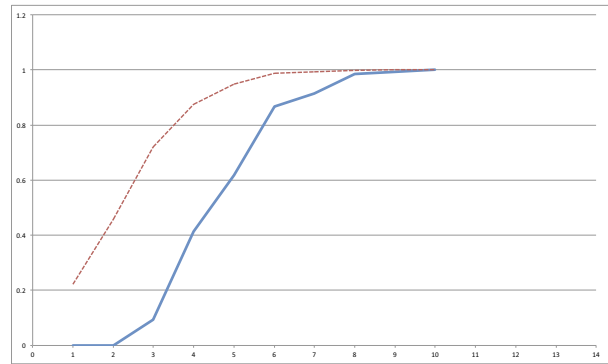
Figure 13: Average fraction of successfully recovered plaintext bytes (dashed line), and success rate for recovering the full 16-byte plaintext (solid line) of the double-byte bias attack for 4-bit-per-byte encoded plaintexts (based on 128 experiments). The unit of the x-axis is 2^{30} encrypted copies of the plaintext.

'0' to '9' and 'a' to 'f'. The setup was similar to the above experiment for base64 encoded messages: we ran a modified version of Algorithm 5 which takes into account the restricted plaintext space, the probability estimates $\{p_{r,k_1,k_2}\}_{1 \leq r \leq 255, \mathtt{0x00} \leq k_1, k_2 \leq \mathtt{0xFF}}$ was based on the biases from [12], and we used a plaintext consisting of bytes with a byte-value corresponding to the character 'b'. The attack was run 128 times for each of $S = 1 \cdot 2^{30}, \ldots, 10 \cdot 2^{30}$ encrypted copies of the plaintext, and the results can be seen in Figure 13. We note:

- With $S = 3 \cdot 2^{30}$ encrypted copies of the plaintext, significantly more than 50% of the plaintext is correctly recovered on average (more precisely, 72% is recovered correctly on average).

- With $S = 5 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is recovered in more than 50% of the 128 trials.

- With $S = 8 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is recovered in nearly all trials (98%).

5.3 Practical Validation

We tested the success rates of our plaintext recovery algorithms in realistic attack settings involving web servers and browsers that are connected through TLS-secured network links. Here, we report on the results.

5.3.1 Validating the operation of RC4 in TLS

We first experimentally verified that the OpenSSL implementation of TLS does indeed use RC4 in the way

explained in Section 2.2, in particular without discarding any initial keystream bytes. We did this by setting up an OpenSSL version 1.0.1c client and server running in a virtualised environment, making use of `s_client` and `s_server`, generic tools that are available as part of the OpenSSL distribution package. The two virtual machines were running Ubuntu 12.10 and kernel version 3.5.0-17.

5.3.2 Validating the single-byte bias attack

Recall that our single-byte bias attack targets the first 256 bytes of plaintext across multiple TLS sessions or connections with random keys. In order to efficiently generate the large number of ciphertexts needed to test our attack, we again used the `s_client` and `s_server` tools, this time modifying the `s_client` source code to force a session resumption for each TLS packet sent.

Using this approach, we were able to generate around 2^{21} encryptions of a fixed plaintext per hour; with 2^{25} recorded ciphertexts, we obtained results comparable to the simulation of our single-byte bias attack reported in Section 5.1 above. A second possible approach to ensure frequently enough rekeying is to actively interfere with the TLS session after each ciphertext is sent, causing it to fail and be restarted, by injecting a bad TLS packet or by resetting the corresponding TCP connection.

We admit that we do not currently have an automated mechanism for forcing session resumption, e.g., from JavaScript. However, JavaScript running in the browser can trigger the browser to establish a fresh TLS session (with a fresh, random key) after each HTTP connection torn down by the attacker. We estimate that this second approach would be significantly slower than using session resumption because of the additional overhead of running the full TLS Handshake. Thus, even though our double-byte bias attack has higher complexity in terms of its ciphertext requirements than our single-byte bias attack, in practice it could be the more efficient attack in terms of total running time, because it can be executed in a single session (or a small number of sessions).

Furthermore, while the single-byte bias attack successfully recovered fixed plaintext bytes in the initial 256 bytes of the TLS ciphertexts, our subsequent experimentation with modern web browsers revealed that these bytes consisted mostly of less interesting HTTP headers rather than cookies. For this reason, after this basic validation, we switched our experimental focus to the double-byte bias attack.

5.3.3 Validating the double-byte bias attack

The double-byte bias attack does not rely on session resumption or session renegotiation and is hence easier to implement in practice. As our experimental setup for this attack, we used a network comprising three (non-virtualized) nodes: a legitimate web server (`www.abc.com`) that serves 16-byte secure cookies over HTTPS, a malicious web server (`www.evil.com`) serving a malicious JavaScript, and a client running a web browser representing a user. The legitimate and malicious web servers run Apache and PHP. For the client, we experimented with various browsers, including Firefox, Opera and Chrome. The nodes were connected through a 100 Mbps Ethernet link; they were equipped with Intel Core i7 processors with 2.3 GHz cores and 16 GB of RAM. None of our experiments used all available CPU resources, nor saturated the network bandwidth.

In this setup, we let the client visit `https://www.abc.com`. This will result in the legitimate web server sending the client a secure cookie which will be stored by the client's browser. This cookie will be the target of the attack. We then let the client visit `http://www.evil.com` and run the malicious JavaScript served by the malicious web server. Note that the same-origin policy (SOP) implemented by the client's browser will prevent the JavaScript from directly accessing the secure cookie. However, the JavaScript will direct repeated HTTP requests to the legitimate server over TLS (i.e. using HTTPS)[11]. The client's browser will then automatically attach the cookie to each request and thereby repeatedly encrypt the target cookie as required in our attack.

The JavaScript uses `XMLHttpRequest` objects[12] to send the requests. We tested `GET`, `POST`, and `HEAD` requests, but found that `POST` requests gave the best performance (using Firefox). Furthermore, we found that the requests needed to be send in blocks to ensure that the browser stayed responsive and didn't become overloaded.

For all the browsers we tested (Firefox, Chrome, and Opera), we found that the requests generated by the JavaScript resulted in TLS messages containing more than 256 bytes of ciphertext. To keep the target cookie in a fixed position in the TLS message (modulo 256) as needed for the double-byte bias attack, we therefore added padding by manipulating the HTTP headers in the request to bring the encrypted `POST` requests up to exactly 512 bytes. This padding introduces some overhead to the attack. The exact amount and location of padding needed is browser-dependent, since different browsers behave differently in terms of the content and order of HTTP headers included in `POST` requests. In practice, then, the attacker's JavaScript would need to perform some browser fingerprinting before carrying out its attack.

As an alternative method for generating request to the legitimate web server, we tried replacing the JavaScript

code with basic `HTML` code, using `HTML` tags such as `img`, pointing to `https://www.abc.com`. The target cookie was still sent in every request, but we found this approach to be less effective (i.e. slower) than using JavaScript.

For Firefox with 512-byte ciphertexts encrypting padded `XMLHttpRequest POST` requests, we were able to generate 6 million ciphertexts per hour on our network, with each request containing the target cookie in the same position (modulo 256) in the corresponding plaintext. Given that our attack needs on the order of $13 \cdot 2^{30}$ encryptions to recover a 16-byte plaintext with high success probability, we estimate that the running time for the whole attack would be on the order of 2000 hours using our experimental setup. The attack generates large volumes of network traffic over long periods of time, and so should not be considered a practical threat. Nevertheless, it demonstrates that our double-byte bias attack does work in principle.

6 Discussion and Conclusions

We have shown that plaintext recovery for RC4 in TLS is possible for the first about 200 or so bytes of the plaintext stream (after the `Finished` message), provided sufficiently many independent encryptions of the same plaintext are available. The number of encryptions required (around 2^{28} to 2^{32} for reliable recovery) is large, but not completely infeasible. We have also shown that plaintext recovery for RC4 is possible from arbitrary positions in the plaintext, given enough encryptions of the same plaintext bytes. Here, the number of encryptions required is rather higher (around $13 \cdot 2^{30}$), but the attack is more flexible and more efficient in practice because it avoids rerunning the TLS Handshake. Certainly, the security level provided by RC4 in TLS is *far* below the strength implied by the 128-bit key in TLS.

This said, it would be incorrect to describe the attacks as being a practical threat to TLS today. However, our attacks are open to further enhancement, using, for example, the ability of our algorithms to output likelihoods for candidate plaintext bytes coupled with more sophisticated plaintext models. It may also be possible to enhance the rate of ciphertext generation in browsers using methods beyond our knowledge. It would seem dangerous to assume that the attacks will not be improved by other researchers in future.

There are countermeasures to the attacks. We discussed these countermeasures extensively with vendors during the disclosure process that we followed prior to making our attacks public. They include: discarding the initial keystream bytes output by RC4, as recommended in [19]; fragmenting the initial HTTP requests at the browser so that the initial keystream bytes are mostly (or entirely) used to encrypt MAC fields; adding random padding to HTTP requests; and limiting the lifetime of cookies or the number of times they can be sent from the browser. The first countermeasure cannot easily be implemented in TLS because it would require mass coordination between the many different client and server implementations. The first two countermeasures are not effective against our double-byte bias attack. The third countermeasure can be relatively easily implemented in browsers but increases the complexity of our attacks rather than defeating them completely. The fourth countermeasure is currently effective, but not immune to further improvements of our attacks. Some vendors (e.g. Opera[13]) have implemented a combination of these (and other) countermeasures; others (e.g. Google in Chrome) are focussing on implementing TLS 1.2 and AES-GCM.

We recognise that, with around 50% of TLS traffic currently using RC4, recommending that it be avoided completely in TLS is not a suggestion to be made lightly. Nevertheless, given the rather small security margin provided by RC4 against our attacks, our recommendation is that RC4 should henceforth be avoided in TLS, and deprecated as soon as possible.

Acknowledgements

References

[1] Wireless LAN medium access control (MAC) and physical layer (PHY) specification, 1997.

[2] Wireless LAN medium access control (MAC) and physical layer (PHY) specification: Amendment 6: Medium access control (MAC) security enhancements, 2004.

[3] ALFARDAN, N., AND PATERSON, K. G. Lucky 13: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy* (2013).

[4] 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. Information Security Group at Royal Holloway, University of London, 2013. `http://www.isg.rhul.ac.uk/tls/RC4biases.pdf`.

[5] AMMAN, B. Personal communication, February 2013.

[6] CANVEL, B., HILTGEN, A., VAUDENAY, S., AND VUAGNOUX, M. Password interception in a SSL/TLS channel. *Advances in Cryptology-CRYPTO 2003* (2003), 583–599.

[7] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999.

[8] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, Apr. 2006.

[9] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, Aug. 2008.

[10] DUONG, T., AND RIZZO, J. Here come the $\oplus$ Ninjas. Unpublished manuscript, 2011.

[11] FLUHRER, S. R., MANTIN, I., AND SHAMIR, A. Weaknesses in the key scheduling algorithm of RC4. In *Selected Areas in Cryptography* (2001), S. Vaudenay and A. M. Youssef, Eds., vol. 2259 of *Lecture Notes in Computer Science*, Springer, pp. 1–24.

[12] FLUHRER, S. R., AND MCGREW, D. Statistical analysis of the alleged RC4 keystream generator. In *FSE* (2000), B. Schneier, Ed., vol. 1978 of *Lecture Notes in Computer Science*, Springer, pp. 19–30.

[13] ISOBE, T., OHIGASHI, T., WATANABE, Y., AND MORII, M. Full plaintext recovery attack on broadcast RC4. In *Preproceedings of FSE* (2013).

[14] JAGANATHAN, K., ZHU, L., AND BREZAK, J. The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows. RFC 4757 (Informational), Dec. 2006.

[15] MAITRA, S., PAUL, G., AND SENGUPTA, S. Attack on broadcast RC4 revisited. In *FSE* (2011), A. Joux, Ed., vol. 6733 of *Lecture Notes in Computer Science*, Springer, pp. 199–217.

[16] MANTIN, I. Predicting and distinguishing attacks on rc4 keystream generator. In *EUROCRYPT* (2005), R. Cramer, Ed., vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 491–506.

[17] MANTIN, I., AND SHAMIR, A. A practical attack on broadcast RC4. In *FSE* (2001), M. Matsui, Ed., vol. 2355 of *Lecture Notes in Computer Science*, Springer, pp. 152–164.

[18] MCGREW, D., AND BAILEY, D. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655 (Proposed Standard), 2012.

[19] MIRONOV, I. (Not so) random shuffles of RC4. In *CRYPTO* (2002), M. Yung, Ed., vol. 2442 of *Lecture Notes in Computer Science*, Springer, pp. 304–319.

[20] PHP DOCUMENTATION GROUP. PHP manual, Feb 2013. http://www.php.net/manual/en/session.configuration.php#ini.session.hash-bits-per-character.

[21] SALOWEY, J., CHOUDHURY, A., AND MCGREW, D. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), Aug. 2008.

[22] SEN GUPTA, S., MAITRA, S., PAUL, G., AND SARKAR, S. Proof of empirical RC4 biases and new key correlations. In *Selected Areas in Cryptography* (2011), pp. 151–168.

[23] SEN GUPTA, S., MAITRA, S., PAUL, G., AND SARKAR, S. (Non-) random sequences from (non-) random permutations – analysis of RC4 stream cipher. *Journal of Cryptology to appear* (2013).

[24] SEPEHRDAD, P., VAUDENAY, S., AND VUAGNOUX, M. Discovery and exploitation of new biases in RC4. In *Selected Areas in Cryptography* (2010), A. Biryukov, G. Gong, and D. R. Stinson, Eds., vol. 6544 of *Lecture Notes in Computer Science*, Springer, pp. 74–91.

[25] SEPEHRDAD, P., VAUDENAY, S., AND VUAGNOUX, M. Statistical attack on RC4 – distinguishing WPA. In *EUROCRYPT* (2011), K. G. Paterson, Ed., vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 343–363.

[26] VAUDENAY, S., AND VUAGNOUX, M. Passive-only key recovery attacks on RC4. In *Selected Areas in Cryptography* (2007), C. M. Adams, A. Miri, and M. J. Wiener, Eds., vol. 4876 of *Lecture Notes in Computer Science*, Springer, pp. 344–359.

Notes

[1]The research of the third, fourth and fifth authors was supported by an EPSRC Leadership Fellowship, EP/H005455/1. The research of the second author was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005.

[2]SSL Pulse (https://www.trustworthyinternet.org/ssl-pulse/) reported in June 2013 that only 15.1% of 170,000 websites surveyed support TLS 1.2; most major browsers currently do not support TLS 1.2.

[3]For examples of RC4 being recommended in the face of CBC attacks, see advice at Qualys' website https://community.qualys.com/blogs/securitylabs/2011/10/17/mitigating-the-beast-attack-on-tls, Ivan Ristic's personal blog http://blog.ivanristic.com/2009/08/is-rc4-safe-for-use-in-ssl.html, PhoneFactor's blog http://blog.phonefactor.com/2011/09/23/slaying-beast-mitigating-the-latest-ssltls-vulnerability, and F5's suggested Lucky 13 mitigation at http://support.f5.com/kb/en-us/solutions/public/14000/100/sol14190.html. Other examples abound on discussion forums and vendor websites.

[4]http://notary.icsi.berkeley.edu

[5]Unfortunately, we do not currently know of a way to trigger TLS session resumption from JavaScript running in a browser.

[6]The Internet Message Access Protocol (IMAP) is a popular protocol for email retrieval.

[7]Note that when attacking secret URL parameters from HTTPS connections or passwords from IMAP sessions such limitations do not arise.

[8] In detail, instead of an internal permutation S of 8-bit values, Fluhrer and McGrew consider variants of RC4 based on permutations of 3-bit, 4-bit, and 5-bit values, respectively. Note that in these versions of RC4, the internal variables i and j, as well as the output Z_r, will also be 3-bit, 4-bit and 5-bit values, respectively.

[9]Note that the internal state S, which corresponds to a permutation over byte values, will not be distributed as a random permutation immediately after the key scheduling algorithm is run, even if the used key is picked uniformly at random. Furthermore, j will not be random, but initialized to 0. However, random S and j will be a close approximation after keystream bytes have been generated a short period of time (see [12] for further discussion of this property).

[10]Our experiments showed that there is no significant difference in the recovery rate when running the attack on encryptions of the plaintext generated by a single key and encryptions generated by a small number of different keys.

[11]This is made possible by Cross-Origin Resource Sharing (CORS), a mechanism developed to allow JavaScript to make requests to another domain than the domain the script originates from.

[12]http://www.w3.org/TR/XMLHttpRequest/

[13]http://my.opera.com/securitygroup/blog/2013/03/20/on-the-precariousness-of-rc4