Statistical Attacks on Cookie Masking for RC4

Kenneth G. Paterson[1] and Jacob C.N. Schuldt[2]

[1] Royal Holloway, University of London, UK
[2] AIST, Japan

Abstract. Levillain *et al.* (AsiaCCS 2015) proposed two cookie masking methods, TLS Scramble and MCookies, to counter a class of attacks on SSL/TLS in which the attacker is able to exploit its ability to obtain many encryptions of a target HTTP cookie. In particular, the masking methods potentially make it viable to continue to use the RC4 algorithm in SSL/TLS. In this paper, we provide a detailed analysis of TLS Scramble and MCookies when used in conjunction with RC4 in SSL/TLS. We show that, in fact, both are vulnerable to variants of the known attacks against RC4 in SSL/TLS exploiting the Mantin biases (Mantin, EUROCRYPT 2005):

- For the TLS Scramble mechanism, we provide a detailed statistical analysis coupled with extensive simulations that show that about 2^{37} encryptions of the cookie are sufficient to enable its recovery.
- For the MCookies mechanism, our analysis is made more complex by the presence of a Base64 encoding step in the mechanism, which (unintentionally) acts like a classical block cipher S-box in the masking process. Despite this, we are able to develop a maximum likelihood analysis which provides a rigorous statistical procedure for estimating the unknown cookie. Based on simulations, we estimate that 2^{45} encryptions of the cookie are sufficient to enable its recovery.

Taken together, our analyses show that the cookie masking mechanisms as proposed by Levillain *et al.* only moderately increase the security of RC4 in SSL/TLS.

Keywords: RC4 stream cipher; statistical analysis; masking.

1 Introduction

The RC4 stream cipher was designed by Rivest in the 1980s. Due to the scarcity of good alternatives, its compact code size, and its good performance in software, the algorithm was widely adopted in practice, including in WEP, WPA/TKIP and SSL/TLS. Indeed, in mid 2013, AlFardan *et al.* [1] reported that RC4 was used in roughly 50% of all SSL/TLS connections. Also in 2013, the first serious cryptanalyses of RC4 in TLS were published [1, 7, 13], with the attacks focussing on the recovery of often-repeated plaintext values in SSL/TLS connections (for example HTTP session cookies or user passwords), using known and newly discovered biases in the RC4 keystreams. Follow-up works [5, 21, 4] substantially reduced the numbers of ciphertexts needed to recover passwords

and HTTP session cookies, by using more powerful biases and better statistical techniques, with the result being that the IETF eventually deprecated the use of RC4 in SSL/TLS [17]. Since the removal of RC4 in SSL/TLS from the main web browsers beginning in early 2016, the amount of RC4-protected traffic for this particular protocol has declined steeply. Attacks against the use of RC4 in WPA/TKIP can be found in [15, 14, 21], and detailed mathematical treatments of empirically discovered biases in RC4 can be found in [6, 19].

One significant effort at remediating attacks targeting HTTP session cookies in SSL/TLS was made by Levillain *et al.* [10]. Their main idea was to *mask* the HTTP session cookie s, roughly speaking sending a randomly generated mask m immediately followed by the XOR of the mask with the HTTP session cookie $s \oplus m$ in place of the raw HTTP session cookie. This masking step removes the repeated plaintext needed for the known attacks against RC4 to succeed. In fact, this repeated plaintext requirement is common to a number of attacks against SSL/TLS, and the protection of RC4 was not the sole focus of [10]. Levillain *et al.* proposed two separate masking mechanisms, *TLS Scramble* and *MCookies*. These differ in the low-level implementation details but both follow the masking principle. Details of the two methods are given in Section 3.

Given the widely-deployed base of RC4 software, and the long-standing attractiveness of the algorithm to software developers, any attempt to resuscitate the algorithm for use in SSL/TLS is deserving of a thorough analysis. Moreover, the masking techniques of Levillain *et al.* are quite simple to understand and would be relatively easy for software engineers to deploy in practice, possibly easier than moving to a later version of TLS supporting more modern algorithms (e.g. AES-GCM in TLS 1.2). There is anecdotal evidence that RC4 is still in quite widespread use in legacy systems and data centres. In our opinion, this makes the cookie masking methods of [10] worthy of detailed examination when used in combination with RC4, even if RC4 is no longer considered a legitimate option for SSL/TLS in modern web browsers.[3]

In this paper, we provide such a study of the two masking countermeasures proposed by Levillain *et al.* [10]. We show that neither provides a great deal of additional security compared to the previous generations of attack on RC4 .

In particular, we show that the TLS Scramble mechanism is vulnerable to a variant of the attacks of [21, 4] exploiting the Mantin biases [11] in RC4. The Mantin biases informally mean that patterns of the form $AB\mathcal{S}AB$ occur slightly more frequently than expected in RC4 keystreams. Here, A and B are byte values, while $\mathcal{S}$ denotes a byte string, and the strength of the bias depends inversely on the length of $\mathcal{S}$. Put another way, if one XORs an RC4 keystream with itself at a particular shift of κ bytes, then the result in consecutive byte positions is more likely to equal $\texttt{0x00}, \texttt{0x00}$ than any other pattern.

If we set κ to be the length of the mask m in the TLS Scramble mechanism, and then consider the XOR of a ciphertext corresponding to the plaintext $m||s \oplus m$ with itself at a shift of κ byte positions, we see that the mask m is can-

[3] And even though the authors of [10] themselves recommend phasing out RC4 rather than relying on their masking countermeasure.

celled out, while, because of the Mantin bias, so too is the keystream (at least, statistically speaking). Thus we will be able to recover the original plaintext value s. The actual attack on TLS Scramble is a little more involved than this, because in reality in the attack, we recover candidate pairs of plaintext bytes in consecutive positions $(i, i+1)$, along with likelihood information for each candidate pair, and our task is to stitch together the overlapping pairs to create high likelihood candidates for s over multiple byte positions. However, the techniques introduced in [21, 4] are readily adapted to handle this task: these techniques use dynamic programming (principally, the list Viterbi algorithm [20]) to produce lists of candidates over multiple bytes given as inputs likelihood information for pairs of candidates on pairs of positions.

We provide a statistical analysis of the main part of our attack on TLS Scramble following the general approach introduced in [4]. This involves computing a log-likelihood function for a plaintext parameter corresponding to a pair of adjacent bytes from the cookie, and then showing how this likelihood can be maximised by evaluating a simple counting function on corresponding ciphertext bytes. We are able to approximate the distributions of the ciphertext counts, and, using theory of order statistics, compute the median rank of the log-likelihood of the true plaintext parameter as a function of the number of available ciphertexts, N. This enables us to make a prediction about the value of N needed for a successful attack in recovering each pair of cookie bytes. To complete the picture, we then give an experimental analysis of our attack on TLS Scramble when targeting complete HTTP cookies (rather than pairs of cookie bytes). The upshot is that, for TLS Scramble, an HTTP session cookie can be recovered using approximately 2^{37} ciphertexts. This figure is not that much greater than the 2^{30} – 2^{32} ciphertexts needed in previous attacks against RC4 in SSL/TLS [21, 4].

Our attack for the MCookies mechanism is more complex, mainly because this second mechanism involves an additional Base64 encoding step being applied separately to the mask and the masked cookie prior to encryption with RC4. This encoding step maps 6-bit blocks of raw binary data to 8-bit printable ASCII values. Because of this extra encoding step, our XOR trick above no longer works directly. However, we still wish to XOR ciphertexts with themselves at fixed shifts to statistically remove the RC4 keystreams (recalling how the Mantin bias effects this on our behalf). We are then forced to deal with the XOR-differential properties of the Base64 encoding map. It transpires that the Base64 encoding negatively affects the statistical effectiveness of our attack and increases its ciphertext requirements, but not too badly. At a high level, this is because Base64 encoding, when considered as a 6-bit to 8-bit S-Box, has less than ideal differential properties.

We are able to obtain a closed-form expression giving the log-likelihoods of pairs of 6-bit plaintext values from the original cookie in terms of a scaled count over ciphertext bytes, with the scale factors being derived from differential properties of the Base64 encoding. Maximising this expression immediately leads to an efficient attack recovering adjacent pairs of 6-bit plaintexts. We are then able

to compute the approximate distribution of our log-likelihood statistic, and using this, develop a thorough understanding of which cookie values are likely to be difficult to recover in our attack, and why. In short, the difficult cookie values are the ones for which the corresponding rows in the difference table for the Base64 encoding take on many small values. This cookie-dependent attack behaviour is in contrast to previous attacks in the literature. Finally, we use an experimental analysis to judge the effectiveness of our attacks. We find that around 2^{45} ciphertexts are sufficient to recover the target cookie with good probability. This is substantially greater than the numbers needed in previous attacks [21, 4] and for our attack on the TLS Scramble mechanism, but much worse than the security properties claimed for the MCookies mechanism in [10].

In tandem, our two attacks indicate that the cookie masking mechanisms of Levillain *et al.* [10] only moderately increase the security of RC4 in SSL/TLS. We therefore do not recommend it for adoption.

1.1 Paper Organisation

In Section 2 we provide more background on the RC4 algorithm and the specific Mantin biases that we will use in our attacks, while in Section 3 we describe the two different masking techniques of [10], TLS Scramble and MCookies. Section 4 describes our attack on the TLS Scramble mechanism, while Section 5 develops our attack on MCookies. In Section 6 we report our experimental results, and in Section 7 we report our conclusions.

2 Background

2.1 The RC4 algorithm

RC4 allows for variable-length key sizes, anywhere from 40 to 256 bits, and consists of two algorithms, namely, a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA). The KSA takes as input an l-byte key and produces the initial internal state $st_0 = (i, j, S)$ for the PRGA; S is the canonical representation of a permutation of the numbers from 0 to 255 where the permutation is a function of the l-byte key, and i and j are indices for S. The KSA is specified in Algorithm 1 where K represents the l-byte key array and S the 256-byte state array. Given the internal state st_r, the PRGA will generate a keystream byte Z_{r+1} as specified in Algorithm 2.

For an overview of how RC4 is used in TLS, see [1, 5]. The salient points for our analysis are as follows: in each TLS connection, RC4 is keyed with a 128-bit key that is effectively uniformly random; this key is used throughout the lifetime of a TLS connection, with the single keystream of bytes that it produces being split up into chunks as needed to encrypt plaintext records using a simple byte-by-byte XOR operation.

Algorithm 1: RC4 key scheduling (KSA)

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

Algorithm 2: RC4 keystream generator (PRGA)

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

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

2.2 The Mantin Biases

Many biases in the RC4 algorithm's outputs have been discovered over the years. Amongst the most powerful and useful for cryptanalysis are the Mantin biases [11], as exploited in [13, 21, 4].

The following result is a restatement of Theorem 1 of Mantin [11], concerning the probability of occurrence of byte strings of the form $AB\mathcal{S}AB$ in RC4 outputs. Here A and B represent bytes and $\mathcal{S}$ denotes an arbitrary byte string of a particular length G.

Result 1 *Let $G \geq 0$ be a small integer and let*

$$\delta_G = \frac{e^{(-4-8G)/256}}{256}.$$

Under the assumption that the RC4 state is a random permutation at step r, then

$$\Pr\left[(Z_r, Z_{r+1}) = (Z_{r+G+2}, Z_{r+G+3})\right] = 2^{-16}\left(1 + \delta_G\right).$$

Notice that for small values of G, the value of δ_G is quite close to 2^{-8}, meaning that the probability in the above result is about $2^{-16}(1 + 2^{-8})$, so that the relative size of the bias is about $1/256$. This is rather small but significant for cryptanalysis when many ciphertexts are available.

Another way of thinking of the result is that it states that, for a shift of $G+2$ positions, the XOR of an RC4 keystream with itself in two consecutive positions is equal to $(\texttt{0x00}, \texttt{0x00})$ with frequency approximately $2^{-16} \cdot \delta_G$ larger than the 2^{-16} expected for a truly random sequence. This is under the assumption that the RC4 state is well-approximated by a random permutation at step r.

The approximate correctness of the above result for RC4 with random keys was experimentally confirmed in [11] for values of G up to 64 and for long

keystreams (i.e. large r values). Further confirmation for the same range of G and for relatively short keystreams was provided in [16]. A more detailed investigation of the Mantin biases was conducted in [4]. It was shown there that the biases are absent for a small number of specific (A, B) byte pairs and stronger for other pairs; however, when averaged over the possible (A, B) pairs, the bias was found to arise with roughly the probability predicted by Result 1.

3 Cookie Masking Countermeasures for HTTPS

In [10], Levillain *et al.* proposed two mechanisms, *TLS Scramble* and *MCookies*, for mitigating the known attacks against various versions of TLS. Both mechanisms are described as providing defense-in-depth, and are based on the masking principle. More specifically, Levillain *et al.* observed that a common prerequisite of the known attacks against TLS is the attacker's ability to obtain multiple encryptions of the secret s the attack aims at recovering. Inspired by the countermeasures against side-channel attacks, the authors proposed to mask s with a randomly generated mask m, each time s is sent i.e. as opposed to sending s, the pair $(m, s \oplus m)$ is sent instead. This allows the receiver to easily recover the original s while ensuring that the message $(m, s \oplus m)$ sent under the protection of TLS changes each time s is sent, thereby obstructing the attacks. In the following, we describe the details of how the two mechanisms proposed in [10] apply this principle.

3.1 TLS Scramble

The TLS Scramble mechanism, as the name suggests, is implemented at the transport layer as an extension of TLS. Specifically, TLS Scramble makes clever use of the support in TLS for compression of the data to be sent, and implements masking as an alternative compression algorithm. The compression algorithm is parameterized by a parameter κ denoting the length of the mask used, and is specified in [10] as the following three-step procedure with input a plaintext P:

1. Generate a κ-byte random string m.
2. Repeat and possibly truncate m to create a new string M such that $|M| = |P|$ i.e. the length of M is equal to that of P.
3. Return the string $X = m||(M \oplus P)$, where $||$ denotes concatenation. Note that $|X| = |P| + \kappa$.

The corresponding decompression algorithm is straightforward, and reverses the above process by extracting the first κ bytes from the received string X, reconstructing M from this, and finally obtaining the plaintext P by XORing the $|X| - \kappa$ last bytes of X with M. TLS allows a compression algorithm to return a plaintext record that is at most 1024 bytes larger than the original plaintext, which implies that $\kappa \leq 1024$.

As TLS Scramble replaces the compression mechanism of TLS, protection against the CRIME and TIME [18] attacks is trivially achieved. Note, however,

that attacks similar to CRIME and TIME might still be possible if compression of the plaintext data is done in the application layer. Levillain *et al.* furthermore claim that TLS Scramble provides protection against BEAST, Lucky 13 [2], RC4 biases [1, 7, 13, 5, 21, 4], and POODLE [12] attacks when using the recommended parameter $\kappa \geq 8$.

3.2 MCookies

The second mechanism from [10], MCookies, introduces masking in the application layer by modifying how web servers handle secure HTTP cookies. Specifically, the HTTP protocol allows a web server to define a cookie to be stored by the client, which will then include the stored cookie on subsequent requests to the same origin. This enables the web server to authenticate requests coming from clients who have already authenticated themselves, e.g. by providing the correct login credentials at the beginning of the HTTP session. In this case, cookie recovery is a desirable goal for a potential attacker, as this would allow him to impersonate clients. MCookies aims at strengthening the protection of cookies when the HTTP session is protected by TLS (i.e. HTTPS) by additionally masking the cookies. Concretely, MCookies modifies the HTTP server behaviour as follows:

Cookie definition When the web server receives a set-cookie request from a web application, e.g. for cookie name $SESSID$ and cookie value V, it generates a random mask M such that $|M| = |V|$, and constructs a masked set-cookie header of the form `Set-cookie:`$SESSID = M : M \oplus V$ (as opposed to the normal unmasked header `Set-cookie:`$SESSID = V$).

Cookie restitution and redefinition Upon receiving a request containing $SESSID = X : Y$, the web server sends $SESSID = X \oplus Y$ to the web application, which corresponds to the original, unmasked cookie. In its response to the client, the web application will then either (1) redefine the cookie, which corresponds to the cookie definition above, (2) erase the cookie, in which case the web server will not have to take further action, or (3) leave the cookie unchanged, in which case the web server redefines the cookie V by generating a new mask M' such $|M'| = |V|$, and sets the new cookie value to $M' : M' \oplus V$ as in the cookie definition above.

For maximal compatibility, the characters used in cookies are typically restricted to a subset of the printable ASCII character set. Base64 encoding [8] is recommended for this purpose in RFC 6265 [3]. Since both the mask M and the masked cookie $M \oplus V$ are (unless otherwise processed) raw binary values, the authors of [10] indeed suggest to use a Base64 encoding of these values (see footnote 8, page 231 of [10]). While this might seem like a technical detail less relevant to security, it plays a significant role in the attack against MCookies presented in Section 5.

The MCookies mechanism described above only modifies the server behaviour. Hence, an active attacker capable of either preventing messages from the client

from reaching the server or otherwise ensuring that the server will not respond e.g. by causing a server-side error, will still be able to make the client repeatedly send the same cookie value over and over again, and thereby sidestep the counter measure implemented by MCookies. To address this, Levillain *et al.* propose an extension of MCookies which modifies the behaviour of clients as well. The extension is made possible by introducing a new cookie attribute, `masked`, which the server can use to indicate to the client that the corresponding cookie should be treated as a masked cookie. Upon receiving a cookie with attribute `masked`, a compatible client will respond by masking the cookie as described above i.e. it will generate a mask M and set the cookie value to $M : M \oplus V$, and include the masked cookie in a new header field `Masked-Cookie` in the response to the server. Hence, when no changes are made to the cookie, the server will not have to include the masked cookie when responding to a client request, and can rely on the client to mask the cookie using a new random mask for each new request. For non-compatible clients, the server would have to resort to the MCookies mechanism described above, which does not require the client to be aware of the masking.

Both MCookies and the client-side extension are claimed to provide protection against BEAST, RC4 biases [1, 7, 13, 5, 21, 4], and POODLE [12], whereas the client-side extension is claimed to additionally provide protection against Lucky 13 [2].

4 Attack on TLS Scramble

4.1 Setting for the Attacks

In this section and the next, we will present attacks against the TLS Scramble and MCookies mechanisms when used in combination with SSL/TLS and when the RC4 algorithm is used for encryption in SSL/TLS. We begin by describing the setting for both attacks, before focussing in the remainder of this section on TLS Scramble.

Both attacks are in the broadcast setting i.e. it is assumed that the attacker has access to multiple encryptions of the target plaintext. In our setting, this target is an HTTP cookie. This assumption holds in practice because of the attacker's ability to inject code (e.g. Javascript) into the client's browser. This code starts SSL/TLS connections to the server, into which the browser automatically places the target cookie.

Furthermore, we assume that the position of the targeted cookie and the surrounding plaintext bytes are known. This is true in our setting because of the highly structured and predictable nature of the initial messages in HTTP connections into which the target cookie is placed.

Finally, it is sufficient for an attack to output a *list* of candidate values for the cookie, since at the end of the attack, many candidates can be tried in turn to see if they are accepted by the server as the correct value. For example, in [4], lists containing up to 2^{18} queries are used, while in [21], lists of size 2^{23} were

used in the most powerful attack. This assumption means that an attack can be considered successful whenever the correct cookie is contained in the output list. This makes it significantly easier to obtain a high success rate compared to attacks which are judged successful only if they output a single cookie candidate that is correct.

These attack assumptions are entirely realistic for SSL/TLS-protected HTTP sessions – see [1, 21, 4] for a more detailed discussion.

The attacks are based on using the Mantin bias which, recall, can be phrased as follows: Let z_i denote the ith keystream byte generated using RC4 with a randomly chosen key and let

$$\delta_G = \frac{e^{(-4-8G)/256}}{256}.$$

Then $\Pr[(z_i, z_{i+1}) = (z_{i+2+G}, z_{i+3+G})] = 2^{-16}(1 + \delta_G)$. In other words, the chance of a keystream byte pair repeating has a relative bias of δ_G, where G denotes the number of keystream bytes (or gap) between the pairs.

4.2 Recovering Two Cookie Bytes with a Single Ciphertext

For the remainder of this section, we focus on TLS Scramble.

Recall that TLS Scramble masks the entire plaintext with a mask M constructed by repeated concatenation (and possible truncation) of a randomly chosen κ-byte mask m. Let $m = m_1 \cdots m_\kappa$, where each m_i corresponds to a byte value.

Consider first a single encryption of a byte pair from a target cookie at location i, (v_i, v_{i+1}), and let (c_i, c_{i+1}) denote the corresponding ciphertext byte pair. For notational convenience, we will assume that the mask m is exactly aligned with (v_i, v_{i+1}) i.e. v_i will be masked with m_1 and v_{i+1} will be masked with m_2 (the attack will work equally well when this is not the case). We can now express (c_i, c_{i+1}) as

$$(c_i, c_{i+1}) = (v_i \oplus m_1 \oplus z_i, v_{i+1} \oplus m_2 \oplus z_{i+1})$$

where z_j is the jth RC4 keystream byte. Furthermore, we may assume that the plaintext byte pair $(v_{i-\kappa}, v_{i+1-\kappa})$ at location $i-\kappa$ is known. The ciphertext byte pair $(c_{i-\kappa}, c_{i+1-\kappa})$ corresponding to the encryption of $(v_{i-\kappa}, v_{i+1-\kappa})$ can then be expressed as:

$$(c_{i-\kappa}, c_{i+1-\kappa}) = (v_{i-\kappa} \oplus m_1 \oplus z_{i-\kappa}, v_{i+1-\kappa} \oplus m_2 \oplus z_{i+1-\kappa}).$$

Note that due to the distance of exactly κ between the byte pairs (v_i, v_{i+1}) and $(v_{i-\kappa}, v_{i+1-\kappa})$, the same bytes, m_1 and m_2, are used in the masking. Hence, by XORing the two ciphertext byte pairs and the known plaintext byte pair, we obtain the byte pair

$$\begin{aligned}(x_i,\, x_{i+1}) &= (c_i, c_{i+1}) \oplus (c_{i-\kappa}, c_{i+1-\kappa}) \oplus (v_{i-\kappa}, v_{i+1-\kappa}) \\ &= (v_i \oplus z_i \oplus z_{i-\kappa},\, v_{i+1} \oplus z_{i+1} \oplus z_{i+1-\kappa})\end{aligned} \tag{1}$$

Note that if $(z_i, z_{i+1}) = (z_{i-\kappa}, z_{i+1-\kappa})$, we have that $(x_i, x_{i+1}) = (v_i, v_{i+1})$. Furthermore, due to the Mantin bias, we know that the probability that $(z_i, z_{i+1}) = (z_{i-\kappa}, z_{i+1-\kappa})$ occurs has a relative bias of $\delta_{\kappa-2}$. In other words, (x_i, x_{i+1}) will take on the value of the target cookie byte pair (v_i, v_{i+1}) slightly more often than any other value. This makes it possible to recover (v_i, v_{i+1}) in the attack.

More formally, let X denote the random variable taking on values (x_i, x_{i+1}) computed as in equation (1) for encryptions using a random RC4 key. Furthermore, let $\theta = (v_i, v_{i+1})$ be a parameter specifying the target cookie byte pair. Then, from the Mantin bias, it follows that

$$\Pr[X = (x_i, x_{i+1}) \mid \theta] \approx \begin{cases} 2^{-16}(1 + \delta_{\kappa-2}) & \text{if } \theta = (x_i, x_{i+1}), \\ 2^{-16} & \text{otherwise} \end{cases}$$

assuming the absence of other biases. This implies that, given the observation (x_i, x_{i+1}), the likelihood of the parameter θ is given by

$$L[\theta\ ;\ (x_i, x_{i+1})] = \Pr[X = (x_i, x_{i+1}) \mid \theta] \approx \begin{cases} 2^{-16}(1 + \delta_{\kappa-2}) & \text{if } \theta = (x_i, x_{i+1}), \\ 2^{-16} & \text{otherwise.} \end{cases}$$

Applying the method of maximum likelihood estimation would immediately suggest setting $\hat{\theta}$, the maximum likelihood estimator for θ, to (x_i, x_{i+1}) as computed in equation (1).

4.3 Recovering Two Cookie Bytes with Many Ciphertexts

The above analysis only considers a single ciphertext. However, it is assumed that multiple encryptions of the target cookie (and known plaintext in offset positions) are available to the attacker, and we wish to make use of all of these ciphertexts in a likelihood analysis.

Let N denote the number of ciphertexts available and let

$$\boldsymbol{X}_i = \left((x_i^{(1)}, x_{i+1}^{(1)}), \ldots, (x_i^{(N)}, x_{i+1}^{(N)}) \right)$$

denote the vector of byte pairs obtained from these as shown in equation (1). Furthermore, let

$$S(\theta; \boldsymbol{X}_i) = |\{(x_i^{(j)}, x_{i+1}^{(j)}) = \theta \mid 1 \leq j \leq N\}|$$

That is, $S(\theta; \boldsymbol{X}_i)$ counts the number of byte pairs $(x_i^{(j)}, x_{i+1}^{(j)})$ from $\boldsymbol{X}_i$ which are equal to the parameter θ. Extending the above likelihood expression to take into account N ciphertexts instead of a single one, we obtain:

$$L[\theta\ ;\ \boldsymbol{X}_i] = \Pr[\boldsymbol{X}_i \mid \theta] = \prod_j \Pr[X^j = (x_i^{(j)}, x_{i+1}^{(j)}) \mid \theta] \approx 2^{-16N}(1 + \delta_{\kappa-2})^{S(\theta; \boldsymbol{X}_i)}$$

where the last equality follows by noting that, for the $N - S(\theta; \boldsymbol{X}_i)$ cases where $(x_i^{(j)}, x_{i+1}^{(j)}) \neq \theta$, the probability $\Pr[X^j = (x_i^{(j)}, x_{i+1}^{(j)}) \mid \theta]$ is approximately equal

to 2^{-16}, while for the $S(\theta; \boldsymbol{X}_i)$ cases where $(x_i^{(j)}, x_{i+1}^{(j)}) = \theta$, the probability $\Pr[X^j = (x_i^{(j)}, x_{i+1}^{(j)}) \mid \theta]$ is given by $2^{-16}(1 + \delta_{\kappa-2})$.

Now applying the method of maximum likelihood estimation suggests using as a maximum likelihood estimate $\hat{\theta}$ for the unknown cookie byte pair (v_i, v_{i+1}) the value maximising the above likelihood expression. Since maximising an expression is equivalent to maximising its logarithm (to any base, but we work with base 2 throughout), we see that we should set $\hat{\theta}$ to the value maximising the log-likelihood:

$$\mathcal{L}[\theta \ ; \ \boldsymbol{X}_i] := S(\theta; \boldsymbol{X}_i) \log(1 + \delta_{\kappa-2}) - 16N. \tag{2}$$

Since the term $-16N$ and the scale factor $\log(1+\delta_{\kappa-2})$ here are the same for every candidate value, we see that it is sufficient to pick $\hat{\theta}$ that maximises $S(\theta; \boldsymbol{X}_i)$, that is, $\hat{\theta}$ should be set to be the two-byte value that occurs most often in $\boldsymbol{X}_i$.

Hence, a simple attack targeting a single cookie byte pair only, can be implemented by computing the counts $S(\theta; \boldsymbol{X}_i)$ for all 2^{16} possible values of the parameter θ, and simply returning the value of θ having the highest count.

4.4 Evaluation of the Attack Recovering Two Cookie Bytes with Many Ciphertexts

It is intuitively clear that, for large enough N, this procedure will produce the correct result, since the target cookie byte pair will eventually have the highest count with overwhelming probability. In this subsection, we adapt the analysis of [4] to obtain a closed form expression for the success probability of the preceding two-byte recovery attack as a function of N and κ.

Let θ^* denote the true value of the plaintext parameter θ. The random variable $S(\theta^*; \boldsymbol{X}_i)$ is approximately distributed as $\mathrm{Bin}(N, 2^{-16}(1 + \delta_{\kappa-2}))$, while for values $\theta \neq \theta^*$, $S(\theta; \boldsymbol{X}_i)$ is approximately distributed as $\mathrm{Bin}(N, 2^{-16})$. For the attack to be successful in a given instance, it is required that the realisation of the single random variable $S(\theta^*; \boldsymbol{X}_i)$ should exceed realisations of the $2^{16} - 1$ other random variables $S(\theta; \boldsymbol{X}_i)$.

If we write $\mu = N2^{-16}$, then $\mathbf{E}(S(\theta^*; \boldsymbol{X}_i)) = 2^{-16}N(1 + \delta_{\kappa-2}) = \mu(1 + \delta_{\kappa-2})$ and $\mathbf{E}(S(\theta; \boldsymbol{X}_i)) = 2^{-16}N = \mu$ for $\theta \neq \theta^*$, with $\mathrm{Var}(S(\theta; \boldsymbol{X}_i)) \approx 2^{-16}N = \mu$ for all θ (to a very good approximation).

For the values of N and hence $\mu = 2^{-16}N$ of interest to us, these binomial random variables are very well-approximated by normal random variables, and we essentially have

$$\begin{aligned} S(\theta^*; \boldsymbol{X}_i) &\sim \mathrm{N}(\mu(1 + \delta_{\kappa-2}), \mu) \\ \text{and } S(\theta; \boldsymbol{X}_i) &\sim \mathrm{N}(\mu, \mu) \qquad\qquad [\theta \neq \theta^*] \end{aligned}$$

where $\mathrm{N}(\mu, \sigma^2)$ denotes a Normal distribution with mean μ and standard deviation σ. We will treat the 2^{16} random variables $S(\theta; \boldsymbol{X}_i)$ as being independent, but of course they are not, since they must sum to N.

It is now convenient to consider the scaling

$$J(\theta; \boldsymbol{X}_i) = \mu^{-\frac{1}{2}} \left(S(\theta; \boldsymbol{X}_i) - \mu \right).$$

Since $J(\theta; \boldsymbol{X}_i)$ is an affine transformation of the original log-likelihood function, it follows that the value of θ which maximises $J(\theta; \boldsymbol{X}_i)$ is also the maximum likelihood estimate $\hat{\theta}$ of the parameter θ given the known data $\boldsymbol{X}_i$.

It is easy to see that $J(\theta; \boldsymbol{X}_i)$ has a Normal distribution with unit variance in both cases ($\theta = \theta^*$ and $\theta \neq \theta^*$). In fact we have:

$$J(\theta^*; \boldsymbol{X}_i) \sim \mathrm{N}\left(\mu^{\frac{1}{2}} \delta_{\kappa-2}, 1\right) \text{ and } J(\theta; \boldsymbol{X}_i) \sim \mathrm{N}(0,1) \text{ for } \theta \neq \theta^*.$$

Furthermore, we may essentially regard all of these random variables $J(\theta; \boldsymbol{X}_i)$ as being independent since the random variables $S(\theta; \boldsymbol{X}_i)$ are close to being independent.

Thus maximising the function $J(\theta; \boldsymbol{X}_i)$ (which is equivalent to maximising $S(\theta; \boldsymbol{X}_i)$) will be successful in giving the true plaintext parameter θ^* if a a realisation of a normal $\mathrm{N}(\mu^{\frac{1}{2}} \delta_{\kappa-2}, 1)$ random variable (corresponding to $J(\theta^*; \boldsymbol{X}_i)$) exceeds every member of a set $\mathcal{R} = \{J(\theta; \boldsymbol{X}_i) \mid \theta \neq \theta^*\}$ of realisations of $2^{16} - 1 = 65535$ independent standard normal $\mathrm{N}(0,1)$ random variables.

At this point, the analysis of [4, Section 4.3] using order statistics applies directly, with the norm $|\delta|$ replaced by scalar $\delta_{\kappa-2}$. This analysis provides the distribution function of $\mathbf{Rk}(\theta^*)$, the expected rank of the log-likelihood of θ^* amongst the 2^{16} log-likelihoods obtained from all possible two-byte values θ. In particular it shows that the median of $\mathbf{Rk}(\theta^*)$ is given by:

$$\mathrm{Median}\left(\mathbf{Rk}(\theta^*)\right) = 2^{16} \Phi\left(-2^{-8} N^{\frac{1}{2}} \delta_{\kappa-2}\right)$$

where Φ denotes the distribution function of a standard normal $\mathrm{N}(0,1)$ random variable.

Taking $\kappa = 9$, which is the value we will use in our experiments in Section 6 and larger than the minimum value of 8 suggested in [10], gives $\delta_{\kappa-2} = \delta_7 = 0.791 \cdot 2^{-8}$. Hence

$$\mathrm{Median}\left(\mathbf{Rk}(\theta^*)\right) = 2^{16} \Phi\left(-0.791 \cdot 2^{-16} N^{\frac{1}{2}}\right).$$

From this expression, the expected value of the median rank of θ^* can be computed for any number of ciphertexts N. Table 1 shows some values of interest. For example, the last two entries show that $N = 2^{37}$ ciphertexts should be sufficient to ensure that the log-likelihood of θ^* is the largest amongst the 2^{16} possible values, and therefore that the correct pair of plaintext bytes is recovered in the attack.

The number of ciphertexts required for a successful attack here can be contrasted with the corresponding 2-byte recovery attack of [4]; there, low values for the expected median rank could already be achieved with around 2^{32} ciphertexts. The difference is attributable to the attack of [4] being able to use multiple Mantin biases in concert, whereas our attack can only make use of one, at a gap of $G = \kappa - 2$.

N	2^{32}	2^{33}	2^{34}	2^{35}	2^{36}	2^{37}	2^{38}
Median $(\mathbf{Rk}(\theta^*))$	14055	8627	3727	828	51	< 1	$\ll 1$

Table 1: Median rank of maximum likelihood estimate of plaintext parameter for $\kappa = 9$.

4.5 Recovering Full Cookies

The above attack can be repeated for adjacent cookie byte pairs, assuming the relevant plaintext bytes are known, to recover a longer cookie value i.e. assuming plaintext bytes $v_{i-\kappa}, \ldots, v_{i+\alpha-1-\kappa}$ are known, the α target cookie bytes $v_i, \ldots, v_{i+\alpha-1}$ can be correctly recovered with high probability for sufficiently large N by repeatedly running the above attack for each of the non-overlapping cookie byte pairs $(v_i, v_{i+1}), (v_{i+2}, v_{i+3}), \ldots, (v_{i+\alpha-2}, v_{i+\alpha-1})$ (here we assume α is even). However, a more effective attack is possible, and one which can output a list of high likelihood candidates for the complete cookie.

The attack is still based on identifying the candidate with the highest likelihood estimate, but now we will consider the entire cookie byte string $v_i \cdots v_{i+\alpha-1}$. Specifically, consider the parameter $\theta = v_i \cdots v_{i+\alpha-1}$ corresponding to a byte string of length α, and let $\theta_j = (v_j, v_{j+1})$, $i \leq j \leq i + \alpha - 2$, be the byte pair of θ starting at position j. Furthermore, let $\mathcal{L}_j[\theta_j \ ; \ \boldsymbol{X}_j]$ be the log-likelihood estimate for $\theta_j = (v_j, v_{j+1})$ as defined in equation (2) and let $\boldsymbol{X} = (\boldsymbol{X}_j)_{j=i}^{i+\alpha-2}$. We can estimate the log-likelihood of the entire string $\theta = v_i \cdots v_{i+\alpha-1}$ as

$$\mathcal{L}[\theta \ ; \ \boldsymbol{X}] \approx \sum_{j=i}^{i+\alpha-2} \mathcal{L}_j[\theta_j \ ; \ \boldsymbol{X}_j] \, .$$

In other words, the attack should identify the byte string $\hat{\theta} = \hat{v}_i \cdots \hat{v}_{i+\alpha-1}$ corresponding to the chain of overlapping byte pairs

$$(\hat{v}_i, \hat{v}_{i+1}), (\hat{v}_{i+1}, \hat{v}_{i+2}), \ldots, (\hat{v}_{i+\alpha-2}, \hat{v}_{i+\alpha-1})$$

for which the sum of the log-likelihoods for the individual byte pairs is the greatest. A formal justification for selecting this summation formula can be found in [4], but it is intuitively the right metric, and we omit the details here.

While the number of possible chains grows exponentially in the length of the chain, the above problem can be solved efficiently using dynamic programming techniques, such as the Viterbi algorithm. This will produce a single candidate for the length α cookie. Using the list Viterbi algorithm [20] instead, we can find the L largest candidates, with L being a parameter of the algorithm that largely dictates its efficiency. See Section 6 for discussion of our experimental results using this approach.

Lastly, it should be noted that the attack still works if the known plaintext bytes start at position $i - t \cdot \kappa$ or $i + t \cdot \kappa$, for any $t \geq 1$. This allows the attack to recover cookies of length greater than κ. However, as t increases, the success

probability of the attack decreases for a fixed number N of ciphertexts, as the relative Mantin bias $\delta_{t\cdot\kappa-2}$ becomes weaker. An alternative approach would be to use known plaintext to recover the first κ cookie bytes, then use those recovered bytes as known plaintext to recover the next group of κ cookie bytes, and so on.

5 Attack on MCookies

In MCookies, only the cookie itself is masked. More specifically, a cookie value V is replaced with a string $M\texttt{:}V \oplus M$ where M is a randomly generated mask of length $|V|$. If this string was directly encrypted using TLS/RC4, the TLS Scramble attack described above would work equally well for MCookies. However, as highlighted in Section 3.2, since M, and therefore $V \oplus M$, are binary strings, these are Base64 encoded to ensure that a "web-safe" character set is used for the replaced cookie value. Recall that Base64 encoding will encode a binary string using the following fixed array of 64 ASCII characters:

$$\begin{aligned}\texttt{base64[]} = [&A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R,\\ &S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l,\\ &m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, /]\end{aligned}$$

Concretely, a binary string $s = s_1 \cdots s_\alpha$, where s_i represents a 6-bit substring[4], is encoded as $\hat{s} = \hat{s}_1 \cdots \hat{s}_\alpha$, where $\hat{s}_i = \texttt{base64}[s_i]$ and s_i is interpreted as an array index (i.e. integer $0 \leq s < 64$). Note that while s is a $(6 \cdot \alpha)$-bit string, $\hat{s}$ is an $(8 \cdot \alpha)$-bit string, as each ASCII character in `base64` is represented by its byte value.

Hence, given a random mask $M = m_1 \cdots m_\alpha$ and a cookie $V = v_1 \cdots v_\alpha$ (where m_i and v_i represent 6-bit strings), the MCookies encoding scheme will result in the plaintext string:

$$\texttt{base64}[m_1] \cdots \texttt{base64}[m_\alpha]\ \texttt{:}\ \texttt{base64}[v_1 \oplus m_1] \cdots \texttt{base64}[v_\alpha \oplus m_\alpha]$$

consisting of $2\alpha + 1$ bytes.

Let $c_1 \cdots c_{2\alpha+1}$ denote the TLS/RC4 encryption of this string, where each c_i corresponds to a ciphertext byte (here we begin our indices at 1 for notational convenience, but the encryption of the encoded cookie need not begin at byte position 1 of the RC4 keystream and all indices can be adjusted accordingly). Similarly to the TLS Scramble attack, we will for $1 \leq i < \alpha$ consider the byte pair

$$(x_i, x_{i+1}) = (c_i, c_{i+1}) \oplus (c_{i+\alpha+1}, c_{i+\alpha+2})$$

formed by XORing pairs of adjacent bytes in the ciphertext at a shift of $\alpha + 1$ positions (note the additional ':' character between M and V adds one to the shift). To ease the notation, we will in the following set

$$\texttt{Xb64}[v_j, m_j] := \texttt{base64}[v_j \oplus m_j] \oplus \texttt{base64}[m_j]$$

[4] For simplicity, we assume the bit-length of s is divisible by 6. If this is not the case, padding rules apply, see [8].

formed by XORing pairs of adjacent bytes in the output of the MCookies encoding scheme at a shift of $\alpha + 1$ positions. We will sometimes leave out the mask m_j when $\texttt{Xb64}[v_j, m_j]$ appears in probability expressions involving a randomly chosen m_j. Using this notation, we have that

$$(x_i, x_{i+1}) = (\texttt{Xb64}[v_i, m_i] \oplus z_i \oplus z_{i+\alpha+1}, \texttt{Xb64}[v_{i+1}, m_{i+1}] \oplus z_{i+1} \oplus z_{i+\alpha+2}).$$

Hence if $(z_i, z_{i+1}) = (z_{i+\alpha+1}, z_{i+\alpha+2})$ occurs (as is the case for the Mantin bias), then we simply have

$$(x_i, x_{i+1}) = (\texttt{Xb64}[v_i, m_i], \texttt{Xb64}[v_{i+1}, m_{i+1}]).$$

Thus we might hope to be able to recover the two 6-bit values (v_i, v_{i+1}) from the cookie in the same way as we did for the TLS Scramble mechanism, by analysing many ciphertexts and selecting as (v_i, v_{i+1}) the pair maximising an appropriate count of occurrences derived from a likelihood estimate.

Unfortunately, the value $\texttt{Xb64}[v, m]$ is not unique for a fixed cookie value $v \neq 0$ when randomly choosing a mask m, so this idea does not work directly. However, the distribution of $\texttt{Xb64}[v, m]$ when considering all choices of m is highly dependent on v. This dependence is illustrated in Figure 2, which shows the distribution of $\texttt{Xb64}[v, m]$ for all possible 6-bit values v, with the colouring indicating the number of times a given 8-bit output arises over the 6-bit values m. (Note that the byte values of all characters in the array `base64[]` are less than 123, and hence all outputs of $\texttt{Xb64}[v, m]$ will be less than 128.)

Hence, given sufficient samples of $\texttt{Xb64}[v, m]$ for a fixed v and randomly chosen m, an attacker would be able to identify v with high probability, by appropriately matching the observed distribution with the known distribution of $\texttt{Xb64}[v, m]$ for random m.

In our situation, we have many observations (x_i, x_{i+1}), which in turn yield pairs $(\texttt{Xb64}[v_i, m_i], \texttt{Xb64}[v_{i+1}, m_{i+1}])$ whenever the Mantin bias holds (for gap $G = \alpha - 1$ with relative bias $2^{-16}\delta_{\alpha-1}$), but which are otherwise essentially randomised. Thus in our attack, we effectively obtain a very noisy version of the distribution of $(\texttt{Xb64}[v_i, m_i], \texttt{Xb64}[v_{i+1}, m_{i+1}])$ over random (m_i, m_{i+1}). From this, we can hope to infer (v_i, v_{i+1}), given sufficiently many ciphertexts.

We now turn to formally describing a maximum likelihood analysis for the problem of selecting (v_i, v_{i+1}) given a collection of ciphertexts and corresponding observations of (x_i, x_{i+1}). This will lead to a concrete procedure for selecting pairs (v_i, v_{i+1}) (along with a likelihood estimate for each such pair).

5.1 Likelihood Analysis of MCookies for a Single Ciphertext

In the following we derive the likelihood of values $\theta = (v_i, v_{i+1})$ given observed data (x_i, x_{i+1}) coming from a single ciphertext. We later extend this to the consideration of multiple ciphertexts. We will let capital letters denote value pairs (i.e. $X_i = (x_i, x_{i+1})$ and $Z_i = (z_i, z_{i+1})$), and use the natural extension

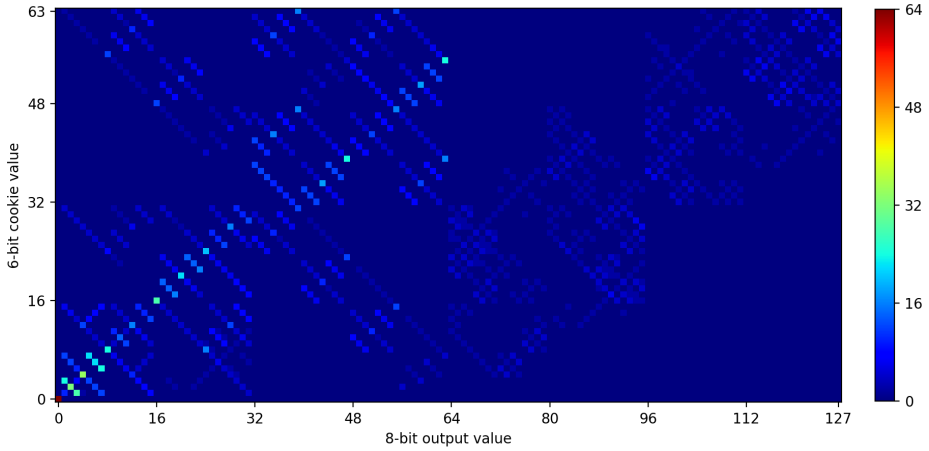

Fig. 2: Heatmap illustrating the distribution of $\texttt{Xb64}[v, m]$ (horizontal axis) for all possible 6-bit values v (vertical axis), with the colouring indicating the number of times a given 8-bit output arises over the 6-bit values m.

$\texttt{Xb64}[\theta] = (\texttt{Xb64}[v_i], \texttt{Xb64}[v_{i+1}])$ (here, omitting the masks (m_i, m_{i+1}) as done in the probability expressions below). We have:

$$\begin{aligned}
L[\theta\ ;\ X_i] &= \Pr[X_i \mid \theta] \\
&= \Pr[\texttt{Xb64}[\theta] \oplus Z_i \oplus Z_{i+\alpha+1} = X_i \mid \theta] \\
&= \sum_{Y \in (\{0,1\}^8)^2} \Pr[(\texttt{Xb64}[\theta] = Y) \wedge (Z_i \oplus Z_{i+\alpha+1} \oplus Y = X_i) \mid \theta] \qquad (3) \\
&= \sum_{Y \in (\{0,1\}^8)^2} \Pr[\texttt{Xb64}[\theta] = Y \mid \theta] \cdot \Pr[Z_i \oplus Z_{i+\alpha+1} \oplus Y = X_i \mid \theta] \qquad (4) \\
&\approx \Pr[\texttt{Xb64}[\theta] = X_i \mid \theta] \cdot 2^{-16}(1 + \delta_{\alpha-1}) \\
&\qquad + (1 - \Pr[\texttt{Xb64}[\theta] = X_i \mid \theta]) \cdot 2^{-16} \qquad (5) \\
&= 2^{-16} \cdot (1 + \Pr[\texttt{Xb64}[\theta] = X_i \mid \theta] \cdot \delta_{\alpha-1}) \qquad (6)
\end{aligned}$$

In the above, equation (3) introduces an auxiliary variable Y, while equation (4) follows from the independence of the choice of mask and the RC4 keystream generation. Equation (5) follows from splitting the sum over Y into two cases: where $Y = X_i$ and where $Y \neq X_i$. In the former case, we have

$$\Pr[Z_i \oplus Z_{i+\alpha+1} \oplus Y = X_i \mid \theta] = \Pr[Z_i \oplus Z_{i+\alpha+1} = (\texttt{0x00}, \texttt{0x00}) \mid \theta],$$

which, from the Mantin bias, is equal to $2^{-16}(1 + \delta_{\alpha-1})$. In the latter case, we have

$$\Pr[Z_i \oplus Z_{i+\alpha+1} \oplus Y = X_i \mid \theta] = \Pr[Z_i \oplus Z_{i+\alpha+1} = W \mid \theta]$$

where $W \neq (\texttt{0x00}, \texttt{0x00})$. Here, the probability is approximately 2^{-16} (in the assumed absence of any keystream bias). Finally, equation (6) follows from some simple algebraic manipulation.

Thus we arrive at:

$$L[\theta\ ;\ X_i] \approx 2^{-16} \cdot (1 + \Pr[\texttt{Xb64}[\theta] = X_i \mid \theta] \cdot \delta_{\alpha-1})$$

as an expression for the likelihood of parameter θ (which, recall, is a possible value of the 6-bit pair of values (v_i, v_{i+1}) from the cookie).

So, given a single ciphertext, the maximum likelihood method would simply dictate selecting as $\hat{\theta}$, the maximum likelihood estimator for θ, the value that maximises the probability $\Pr[\texttt{Xb64}[\theta] = X_i]$. Note that there may be several values θ attaining the same value, because the map $\texttt{Xb64}[\cdot]$ is not injective.

5.2 Likelihood Analysis of MCookies for Multiple Ciphertexts

We now consider a vector of N values $\boldsymbol{X}_i = (X_i^{(1)}, \ldots, X_i^{(N)})$ derived from the ciphertexts of N encryptions of the target cookie. This cookie is originally encoded as 6α bits, which are mapped to α bytes by Base64 encoding and thence to $2\alpha + 1$ bytes by the MCookies mechanism. We still focus on the recovery of two consecutive 6-bit portions (v_i, v_{i+1}) of the target cookie $V = v_1 \ldots v_\alpha$. We let θ denote this value, regarded as a parameter.

We let

$$S(Y; \boldsymbol{X}_i) = |\{X_i^{(j)} = Y \mid 1 \leq j \leq N\}|,$$

i.e. $S(Y; \boldsymbol{X}_i)$ is the count of X_i values equalling a given Y. Extending the previously derived likelihood expression to take into account $\boldsymbol{X}_i$ as opposed to just a single pair X_i, we obtain

$$L[\theta\ ;\ \boldsymbol{X}_i] \approx 2^{-16N} \cdot \prod_{Y \in (\{0,1\}^8)^2} (1 + \Pr[\texttt{Xb64}[\theta] = Y \mid \theta] \cdot \delta_{\alpha-1})^{S(Y; \boldsymbol{X}_i)}$$

yielding a log-likelihood expression:

$$\begin{aligned}\mathcal{L}[\theta; \boldsymbol{X}_i] &\approx -16N + \sum_{Y \in (\{0,1\}^8)^2} S(Y; \boldsymbol{X}_i) \cdot \log(1 + \Pr[\texttt{Xb64}[\theta] = Y \mid \theta] \cdot \delta_{\alpha-1}) \\ &\approx -16N + \sum_{Y \in (\{0,1\}^8)^2} S(Y; \boldsymbol{X}_i) \cdot \Pr[\texttt{Xb64}[\theta] = Y \mid \theta] \cdot \delta_{\alpha-1}.\end{aligned}$$

Since, for each choice of θ, the additive term $-16N$ and the multiplicative factor $\delta_{\alpha-1}$ are the same, we see that in order to determine the maximum likelihood estimator $\hat{\theta}$ we can instead maximise the scaled log-likelihood:

$$\mathcal{SL}[\theta; \boldsymbol{X}_i] \approx \sum_{Y \in (\{0,1\}^8)^2} S(Y; \boldsymbol{X}_i) \cdot \Pr[\texttt{Xb64}[\theta] = Y \mid \theta]. \tag{7}$$

This analysis leads directly to an attack recovering two 6-bit values (v_i, v_{i+1}) from the cookie: simply compute the above sum for each candidate $\theta = (v_i, v_{i+1})$ and output the value $\hat{\theta}$ having the highest value.

5.3 Analysis of the Scaled Log-likelihood

The expression for $\mathcal{SL}[\theta; \boldsymbol{X}_i]$ in equation (7) can be interpreted as saying that, in order to evaluate the (scaled) log-likelihood of a given θ, we take a scaled sum of the 2^{16} counts $S(Y; \boldsymbol{X}_i)$ with the scale factors being the probabilities $\Pr[\texttt{Xb64}[\theta] = Y]$ (where the probability is taken over the suppressed mask value M). Note that the majority of the scale factors will be zero, because the output tables of the function `Xb64` are quite sparse (cf. Figure 2, showing the output behaviour for a single 6-bit input value, whereas we are now considering pairs of 6-bit input values θ).

There are many pairs θ, θ' for which the distributions $\texttt{Xb64}[\theta]$, $\texttt{Xb64}[\theta']$ are quite close and which therefore present "close" sets of scale factors when evaluating the modified log-likelihood expression (7). Thus we can expect such pairs θ, θ' to lead to comparable likelihood values. Since the success of the attack relies on the likelihood of θ^*, the true value of the parameter, exceeding the log-likelihoods of all the other $2^{12}-1$ values θ, it is clear that the attack performance will now depend heavily on θ^*, with poorer performance in those cases where the distribution of $\texttt{Xb64}[\theta^*]$ is close to the distribution for other values θ, and better performance when it is not (equivalently, larger number of ciphertexts will be needed in such cases to get a good success probability for the attack). This behaviour is in contrast to our previous attack on TLS Scramble, and indeed prior attacks in the literature. Moreover, we can expect our maximum likelihood attack on MCookies to require more ciphertexts than our attack on TLS Scramble because of the presence of the `Xb64` map, which effectively reduces the size of the biases involved in making the attack work. Intuition for this can be gleaned from looking just at the likelihood expressions based on a single ciphertext, where the term $2^{-16}(1 + \delta_{\kappa-2})$ is replaced by $2^{-16} \cdot (1 + \Pr[\texttt{Xb64}[\theta] = X_i \mid \theta] \cdot \delta_{\alpha-1})$.

We formalise the above intuitions in the remainder of this section. First, we develop an understanding of the distribution of $S(Y; \boldsymbol{X}_i)$; we then apply this to compute the distribution of $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$, where θ^* is the true value of the plaintext parameter, and of $\mathcal{SL}[\theta; \boldsymbol{X}_i]$ for $\theta \neq \theta^*$.

First, we claim that

$$S(Y; \boldsymbol{X}_i) \sim \mathrm{Bin}(N, 2^{-16}(1 + \delta_{\alpha-1} \cdot \Pr[\texttt{Xb64}[\theta^*] = Y])) \,.$$

To see why, recall that

$$(x_i, x_{i+1}) = (\texttt{Xb64}[v_i, m_i] \oplus z_i \oplus z_{i+\alpha+1}, \texttt{Xb64}[v_{i+1}, m_{i+1}] \oplus z_{i+1} \oplus z_{i+\alpha+2}). \tag{8}$$

Here, (v_i, v_{i+1}) is the true pair of 6-bit cookie values, i.e. the value of θ^*. Alternatively, this can be written in our vector notation as:

$$X_i = \texttt{Xb64}[\theta^*, M] \oplus Z_i \oplus Z_{i+\alpha+1} \,. \tag{9}$$

Hence, a specific value Y arises for X_i from a ciphertext if either $Z_i = Z_{i+\alpha+1}$ (in which case we must have $\texttt{Xb64}[\theta^*, M] = Y$ for some M), or if $Z_i \neq Z_{i+\alpha+1}$ (in which case we have $\texttt{Xb64}[\theta^*, M] = Y'$ for some M and some value of $Y' \neq Y$).

In the former case, we have an event with probability

$$2^{-16}(1+\delta_{\alpha-1})\Pr[\texttt{Xb64}[\theta^*]=Y]$$

because of the Mantin bias. In the latter case, we have an event with probability approximately

$$2^{-16}(1-\Pr[\texttt{Xb64}[\theta^*]=Y]).$$

This can be seen by noting that with X_i fixed and $\texttt{Xb64}[\theta^*, M]$ also fixed to some value $Y' \neq Y$, then given any fixed Z_i, the probability that $Z_{i+\alpha+1}$ provides a solution to equation (9) is approximately 2^{-16}. Here we make the reasonable assumption for RC4 keystreams that $Z_{i+\alpha+1}$ is still uniformly random when conditioned on a fixed Z_i and $Z_{i+\alpha+1} \neq Z_i$.

These two events are exclusive. So, summing the above two probabilities and simplifying, we obtain

$$\Pr[Y=X_i] \approx 2^{-16}(1+\delta_{\alpha-1}\cdot\Pr[\texttt{Xb64}[\theta^*]=Y])\,.$$

Our preceding claim that

$$S(Y;\boldsymbol{X}_i) \sim \mathrm{Bin}(N, 2^{-16}(1+\delta_{\alpha-1}\cdot\Pr[\texttt{Xb64}[\theta^*]=Y]))$$

now follows directly from the definition of $S(Y;\boldsymbol{X}_i)$ and the assumption that the N different ciphertext byte pairs $X_i^{(j)}$ behave independently.

We now use the usual Normal approximation to the binomial distribution (which certainly holds well for the large N that we consider) to approximate the distribution of $S(Y;\boldsymbol{X}_i)$ by $\mathrm{N}(q_{\theta^*,Y}N, 2^{-16}N)$ where

$$q_{\theta^*,Y} = 2^{-16}(1+\delta_{\alpha-1}\cdot\Pr[\texttt{Xb64}[\theta^*]=Y]).$$

Henceforth, we assume that the Normal distributions describing the random variables $S(Y;\boldsymbol{X}_i)$ are independent.

With this computation in hand, we can now evaluate the distribution of $\mathcal{SL}[\theta;\boldsymbol{X}_i]$. Specifically, we use standard results for the mean and variance of scaled sums of independent Normal distributions to deduce that

$$\begin{aligned}\mathcal{SL}[\theta;\boldsymbol{X}_i] &\approx \sum_{Y\in(\{0,1\}^8)^2} S(Y;\boldsymbol{X}_i)\cdot\Pr[\texttt{Xb64}[\theta]=Y]\\ &\sim \sum_{Y\in(\{0,1\}^8)^2} \mathrm{N}(q_{\theta^*,Y}N, 2^{-16}N)\cdot\Pr[\texttt{Xb64}[\theta]=Y]\\ &\sim \mathrm{N}(\mu,\sigma^2)\end{aligned}$$

where

$$\begin{aligned}\mu &= \sum_{Y\in(\{0,1\}^8)^2} q_{\theta^*,Y} N \cdot \Pr[\texttt{Xb64}[\theta] = Y] \\ &= \sum_{Y\in(\{0,1\}^8)^2} 2^{-16} N(1 + \delta_{\alpha-1} \cdot \Pr[\texttt{Xb64}[\theta^*] = Y]) \cdot \Pr[\texttt{Xb64}[\theta] = Y] \\ &= 2^{-16} N \sum_{Y\in(\{0,1\}^8)^2} \Pr[\texttt{Xb64}[\theta] = Y] \\ &\qquad + 2^{-16} N\delta_{\alpha-1} \sum_{Y\in(\{0,1\}^8)^2} \Pr[\texttt{Xb64}[\theta^*] = Y] \cdot \Pr[\texttt{Xb64}[\theta] = Y] \\ &= 2^{-16} N + 2^{-16} N\delta_{\alpha-1} \cdot \Theta(\theta^*, \theta).\end{aligned}$$

Here

$$\Theta(\theta^*, \theta) := \sum_{Y\in(\{0,1\}^8)^2} \Pr[\texttt{Xb64}[\theta^*] = Y] \cdot \Pr[\texttt{Xb64}[\theta] = Y]$$

and then

$$\begin{aligned}\sigma^2 &= \sum_{Y\in(\{0,1\}^8)^2} 2^{-16} N \cdot \Pr[\texttt{Xb64}[\theta] = Y]^2 \\ &= 2^{-16} N \cdot \Theta(\theta, \theta).\end{aligned}$$

In the above expressions for the mean and variance of the approximate distribution for $\mathcal{SL}[\theta; \boldsymbol{X}_i]$, the quantity $\Theta(\theta^*, \theta)$ is formed as an inner product between the two distributions $\Pr[\texttt{Xb64}[\theta^*] = Y]$ and $\Pr[\texttt{Xb64}[\theta] = Y]$. One can also think of it as a scaled version of the inner product between two rows of $\texttt{Xb64}[\cdot]$, the difference map of the 12-bit to 16-bit S-box obtained from the Base64 encoding map, and therefore as a measure of correlation between the two distributions or table rows. Thus $\Theta(\theta, \theta)$ and $\Theta(\theta^*, \theta^*)$ are norms of these rows.

As a particular case of the above computations, where we substitute θ by θ^* we have that

$$\mathcal{SL}[\theta^*; \boldsymbol{X}_i] \sim \mathrm{N}(\mu^*, (\sigma^*)^2)$$

where

$$\mu^* = 2^{-16} N + 2^{-16} N\delta_{\alpha-1} \cdot \Theta(\theta^*, \theta^*) \quad \text{and} \quad (\sigma^*)^2 = 2^{-16} N \cdot \Theta(\theta^*, \theta^*).$$

We see that the difference in means for the distributions $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$ and $\mathcal{SL}[\theta; \boldsymbol{X}_i]$ for arbitrary values of θ is proportional to $\Delta(\theta^*, \theta) := \Theta(\theta^*, \theta^*) - \Theta(\theta^*, \theta)$. It follows that values θ for which the two distributions $\Pr[\texttt{Xb64}[\theta^*] = Y]$ and $\Pr[\texttt{Xb64}[\theta] = Y]$ are highly correlated (so that $\Delta(\theta^*, \theta)$ is small) will be hard to distinguish via the two likelihood functions $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$, $\mathcal{SL}[\theta; \boldsymbol{X}_i]$: their means will tend to be close, and it is therefore probable that $\mathcal{SL}[\theta; \boldsymbol{X}_i]$ will exceed $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$, so that the maximum likelihood estimator will produce the wrong output (i.e. $\hat{\theta} \neq \theta^*$).

Furthermore, if $\Theta(\theta^*, \theta^*)$ is small, then we might expect this effect to be amplified: while the mean of $\mathcal{SL}[\theta; \boldsymbol{X}_i]$ will not exceed that of $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$, the

variance of $\mathcal{SL}[\theta; \boldsymbol{X}_i]$, being proportional to $\Theta(\theta, \theta)$, could significantly exceed that of $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$ for some values of θ. Then there is a fair chance that $\mathcal{SL}[\theta; \boldsymbol{X}_i]$ will exceed $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$ despite having a lower mean. Generally, $\Theta(\theta^*, \theta^*)$ is small when the distribution $\Pr[\texttt{Xb64}[\theta^*] = Y]$ takes many positive (and necessarily small) values. This effect can be quite large: for example, when θ^* consists of 2 6-bit all-zero values, we have $\Pr[\texttt{Xb64}[\theta^*] = (\texttt{0x00}, \texttt{0x00})] = 1$ so that $\Theta(\theta^*, \theta^*) = 1$ (the largest possible value for Θ), but when $\theta^* = (\texttt{0x2d}, \texttt{0x2d})$ we have $\Theta(\theta^*, \theta^*) = 0.00494385$, which is 202 times smaller than the maximum value.

Recall also that the maximum likelihood approach succeeds only if $\mathcal{SL}[\theta^*; \boldsymbol{X}_i]$ exceeds $\mathcal{SL}[\theta; \boldsymbol{X}_i]$ for *every* choice of $\theta \neq \theta^*$, and there are $2^{12} - 1$ such choices where the procedure could fail. Thus the above effects are magnified by the large number of θ values that need to be considered.

Finally, we note that the difference in means is equal to $2^{-16} N \delta_{\alpha-1} \Delta(\theta^*, \theta)$, while the corresponding difference in means for our TLS Scramble attack is equal to $2^{-16} N \delta_{\kappa-2}$. In general, $\delta_{\alpha-1}$ and $\delta_{\kappa-2}$ will be different but of the same order of magnitude. On the other hand, $\Delta(\theta^*, \theta)$ acts as a scaling factor in reducing the difference of means when comparing our two attacks. This gives a heuristic reason for why our MCookies attack will require more ciphertexts than our attack on TLS Scramble: the log-likelihood measures are harder to distinguish for MCookies because their means are closer together.

In summary, we have obtained approximations to the distribution of the likelihood measures $\mathcal{SL}[\theta; \boldsymbol{X}_i]$; their subsequent analysis suggests that certain values for θ^*, the true plaintext parameter, will be difficult to recover with high reliability. Such values correspond to distributions $\Pr[\texttt{Xb64}[\theta^*] = Y]$ that are highly correlated with other distributions $\Pr[\texttt{Xb64}[\theta] = Y]$ and in particular for which $\Theta(\theta^*, \theta^*)$, the norm of the relevant row in the $\texttt{Xb64}[\cdot]$ table, is small.

5.4 Recovering Full Cookies

Finally, we turn from consideration of adjacent pairs of cookie symbols to the problem of recovering full cookies in an attack.

As in TLS Scramble, we can approximate the log-likelihood for the entire cookie value $\theta = v_1 \cdots v_\alpha$ by the sum of the (approximate) log-likelihoods for the overlapping 6-bit value pairs $(v_1, v_2), (v_2, v_3), \ldots (v_{\alpha-1}, v_\alpha)$, i.e.

$$\mathcal{L}[\theta \ ; \ \boldsymbol{X}] \approx \sum_{j=1}^{\alpha-1} \mathcal{L}_j[\theta_j \ ; \ \boldsymbol{X}_j]$$

where

$$\mathcal{L}_j[\theta_j \ ; \ \boldsymbol{X}_j] \approx -16N + \sum_{Y \in (\{0,1\}^8)^2} S(Y; \boldsymbol{X}_j) \cdot \Pr[\texttt{Xb64}[\theta_j] = Y \mid \theta_j] \cdot \delta_{\alpha-1}.$$

and

$$\boldsymbol{X} = (\boldsymbol{X}_j)_{j=1}^{\alpha-1}$$

is an array of size $N \times (\alpha - 1)$ obtained from the available ciphertexts.

Of course, common terms $(-16N)$ and scale-factors $(\delta_{\alpha-1})$ can be removed from the expression for $\mathcal{L}_j$ when computing the summation over j values.

Our attack will then identify the cookie value $\hat{\theta} = \hat{v}_1 \cdots \hat{v}_\alpha$ which maximizes $\mathcal{L}[\theta \ ; \ \boldsymbol{X}]$. As in the TLS Scramble case, this can be done efficiently using dynamic programming techniques. Our experimental results for recovering cookies protected with the MCookies method are presented in the next section.

6 Experimental Results

To validate the attacks presented in Sections 4 and 5 against TLS Scramble and MCookies, respectively, we ran several experiments simulating the attacks, and measured the obtained attack success rates. The details of the simulations as well as the obtained results are described below.

6.1 TLS Scramble

Methodology For the attack against TLS Scramble, we used a mask length of $\kappa = 9$ (Levillain et al. [10] recommends $\kappa \geq 8$), and mainly focused on the recovery of 16 unknown bytes of a Base64 encoded cookie. The ten plaintext bytes preceding and the ten plaintext bytes following the cookie bytes were assumed to be known. This corresponds to a commonly encountered scenario for HTTP cookies, and allows the attack to essentially consider an extended 18 byte cookie with known first and last byte values. In this setting, we considered an attack using the known plaintext bytes preceding the cookie to recover the first eight bytes of the cookie, and the known plaintext bytes following the cookie to recover the last eight bytes of the cookie, effectively running the attack "backwards". Note that the strength of the Mantin bias δ_k, and thereby the attack's success probability, decreases with the distance between the considered byte pairs i.e. considering the closest possible pairs of known plaintext and target cookie bytes is an advantage in the attack. Furthermore, we assumed the two halves of the cookie are recovered independently in the attack. This allow us to derive the success probability of recovering the entire 16-byte cookie from the success probability of recovering the first or last 8-byte half of the cookie (note that, when reordering the ciphertext bytes, it makes no difference whether the first or the last half of the cookie is recovered in the attack, as the Mantin biases are symmetric). Various optimisations of the attack in this setting are possible. It would, for example, be possible to exploit both the preceding and following known plaintext bytes simultaneously in the recovery of cookie bytes. We did not consider these optimisations further.

For the attack, we used the list Viterbi algorithm based on the likelihood estimate derived in Section 4.5 with a list size of $L = 2^{15}$. Due to the properties of the algorithm, this allow us to derive the results for any $L < 2^{15}$. In the attack, we consider the known byte immediately preceding the unknown cookie values to be part of the cookie when constructing the XOR values (x_i, x_{i+1}) defined in

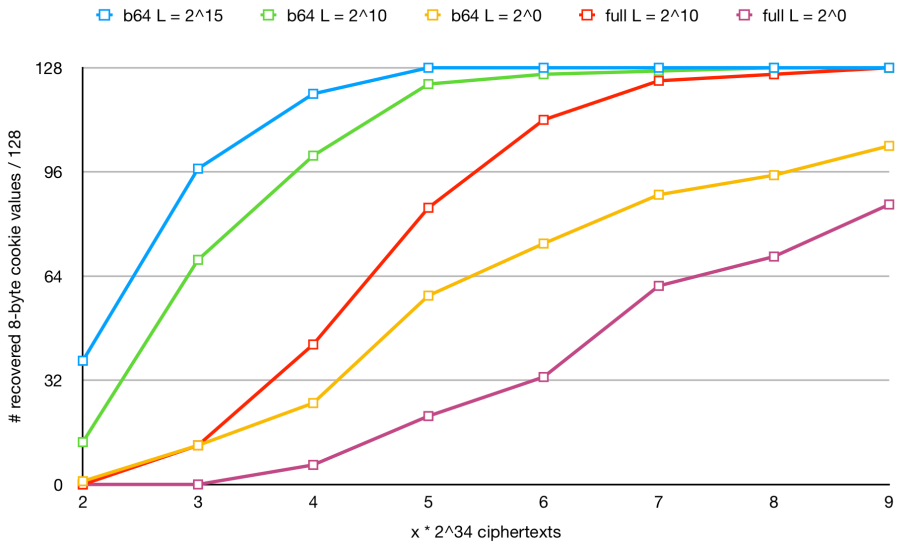

Fig. 3: Measured success rate of the attack against TLS Scramble ($\kappa = 9$) for 8-byte cookie values. The abbreviation `b64` denotes an attack against a Base64 encoded cookie; `full` denotes an attack against a cookie consisting of byte values taken from $\{0, 1\}^8$; L denotes the list size of the list Viterbi algorithm.

Section 4.2, and not part of the known preceding plaintext bytes $(v_{i-\kappa}, v_{i-\kappa+1})$. This ensures that the list Viterbi algorithm, which process the (x_i, x_{i+1}) byte pairs can be initiated with a known first byte value. Note, however, that no terminating byte value is known, as we target the recovery of half cookies.

To measure the success rate of the attack, we generated ciphertexts in sets of size $N = 2^{34}$ and $N = 2^{35}$, and for each set stored the counts $S(\theta, \boldsymbol{X}_i)$ defined in Section 4.3 for all possible values of θ and for the relevant values of i. These were then accumulated to create counts for progressively larger values of N of the form $N = x \cdot 2^{34}$ for $x \in \{2, 3, \ldots, 9\}$. In turn these counts were used in the attack simulation. We generated counts to simulate a total of 128 independent attacks. For completeness, we additionally simulated the attack for binary cookies consisting of 16 randomly chosen bytes (as opposed to Base64 encoded cookies) using a list size of $L = 2^{10}$, which allowed us to additionally derive the corresponding results for $L = 1$.

All computations needed for the simulations were done using a server equipped with a four-core Intel i7 3.4GHz CPU.

Results Figure 3 shows the measured success rate of the attack for the recovery of half of a 16-byte cookie for different values of N, whereas Figure 4 shows the derived success rate for a full 16-byte cookie. The latter simply corresponds to the expected success rate of the recovery of both halves of the cookie, based on

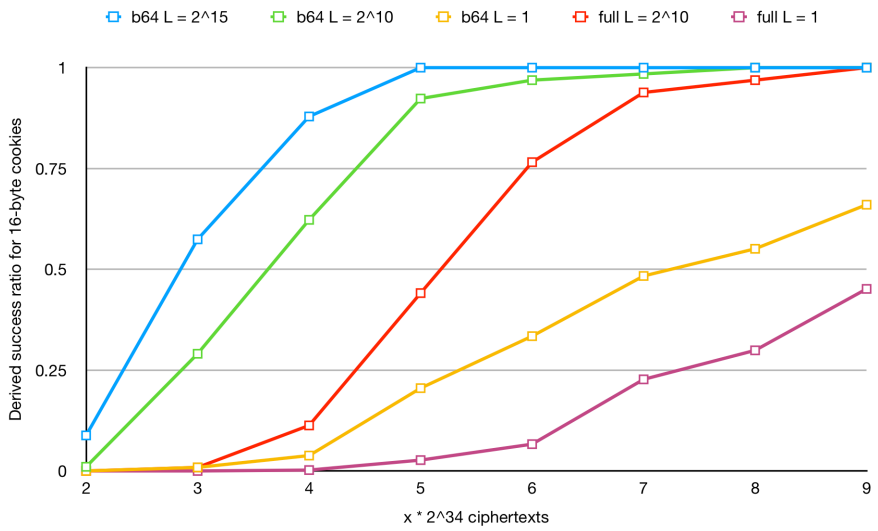

Fig. 4: Derived success rate of attack against TLS Scramble ($\kappa = 9$) for 16-byte cookie values. The abbreviation `b64` denotes an attack against a Base64 encoded cookie; `full` denotes an attack against a cookie consisting of byte values taken from $\{0,1\}^8$; L denotes the list size of the list Viterbi algorithm.

the measurements shown in Figure 3. Each curve represents an attack against either a Base64 encoded cookie (denoted `b64`) or a cookie with byte values from the full byte range $\{0,1\}^8$ (denoted `full`), as well as a list size of either $L = 1$, $L = 2^{10}$, or $L = 2^{15}$.

As expected, Figure 4 shows, for a fixed N, a larger list size L increased the success rate, with the success rate for the attack against a Base64 encoded cookie with list size $L = 2^{15}$ reaching 1 at $N = 5 \cdot 2^{34}$, whereas the corresponding success rate for a list size $L = 1$ is 0.2. Likewise, in the case of a cookie with byte values in the full byte range $\{0,1\}^8$, a success rate of 1 is reached for $N = 9 \cdot 2^{34}$ and $L = 2^{10}$, whereas the corresponding success rate for $L = 1$ is 0.45.

In Section 4 we showed that, once about $N = 2^{37}$ ciphertexts are available, we would expect the attack recovering 2-byte cookie values to succeed with high probability (because the expected value of the median rank of the log-likelihood of θ^* is less than 1). However, our experiments target full cookie recovery while the evaluation in Section 4 was for recovery of just 2 bytes from cookies. This makes our full cookie recovery attack harder to perform with $N = 2^{37}$ than the analysis of Section 4 would suggest. On the other hand, our use of a *list* Viterbi algorithm for large L makes cookie recovery easier (in that we adjudge an attack successful if the cookie appears anywhere in the output list). Despite these differences, the results portrayed in Figures 3 and 4 are broadly consistent with the analysis of Section 4, with good success rates being found in our simulations across the board for $N = 8 \cdot 2^{34} = 2^{37}$.

Cookie	Value (hex)
1	b4 14 4d 19 b0 49 40 72 ba d3 79
2	f9 bd b7 2d 02 fd e0 26 85 2d d8
3	f1 c7 80 cd 25 02 86 53 b4 0f cc
4	c6 b7 42 6f c2 af e5 13 19 2d 17
5	63 17 b3 68 3c a5 fb a8 d8 29 4c
6	0c 66 2c 90 e7 71 38 1a 3c c4 97
7	d2 c1 08 67 96 82 ac ef 51 f8 c3
8	ea 52 f2 b0 1c d0 e9 33 27 81 34
9	78 4d bb af da c7 13 09 03 29 06
10	32 bf 93 82 51 a7 e2 33 67 9e f3
11	1d e0 b1 d0 e3 1b 23 b8 ee 47 16
12	8a e5 15 2d ad b7 39 01 b9 e5 d4
13	2b 66 d7 13 d1 ae da 24 f7 4a 4e
14	30 69 0f 9a e6 19 09 07 7f f7 d3
15	16 41 fb 95 d5 9c 83 39 f0 c7 e0
16	62 19 30 f4 49 6b 5e 8a 8b 57 8b

Fig. 5: Cookie values used in the simulations of the attack on MCookies.

6.2 MCookies

Methodology For the attack against MCookies, we considered a cookie consisting of 11 bytes. This corresponds to a 16-byte cookie value when masked and Base64 encoded, including a single Base64 padding character '='. It results in a 33-byte encoded value when considering the entire MCookies encoding process.

In this setting, a list Viterbi algorithm with known first and terminating byte values can be used due to the HTTP cookie having a known character preceding the mask/cookie value (the character '='), the mask and the masked cookie value being separated by a known character (the character ':'), and the same padding character being used for both the mask and the masked cookie i.e. in the construction of the $\boldsymbol{X}_i$ values described in Section 5.2, we considered the XOR of the encrypted mask including the preceding character '=', and the encrypted masked cookie including the preceding character ':'. We used a list size of $L = 2^{15}$, which allowed us to derived the corresponding results for $L = 1$ and $L = 2^{10}$.

Unlike the TLS Scramble attack, the success rate of the MCookies attack is dependent on the cookie value being recovered (see discussion in Section 5). In the attack simulations, we generated the used cookie values using AES in counter mode. Specifically, we set each byte of the key to the value `0x02`, and the IV to a cookie number which was increased for each new cookie, starting with the value 1. The generated keystream was used as the byte values of the cookie. The generated cookie values used in our experiments are shown in Figure 5.

For each cookie value, we generated ciphertexts in sets of size $N = 2^{35}$, stored the counts $S(Y; \boldsymbol{X}_i)$ defined in Section 5.2 for all possible values Y and the positions $i \in \{0, \ldots, 15\}$ corresponding to the length of the encoded masked cookie including the known preceding character. These counts were accumulated

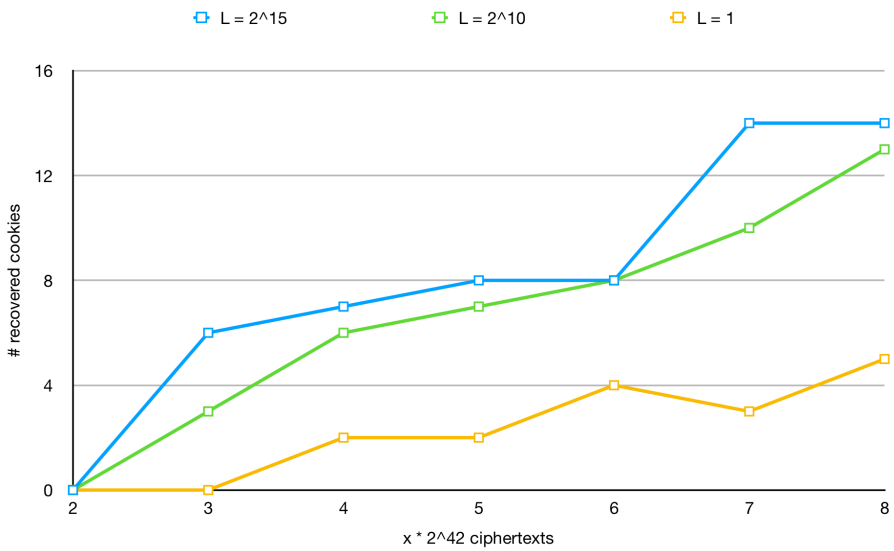

Fig. 6: Measured success rate of attack against MCookies for 16-byte (11-byte unencoded) cookie values. L denotes the list size of the list Viterbi algorithm.

to construct counts for progressively larger N, but as the biases in the case of MCookies are significantly weaker than in the case of TLS Scramble, we considered N of the form $x \cdot 2^{42}$ for $x \in \{2, \ldots, 8\}$. Furthermore, due to the large amount of computational resources required, we were able to generate data sufficient to simulate only 16 independent attacks, each for a different cookie value. This corresponds to data derived from a total of 2^{49} ciphertexts.

The generation of ciphertext data to simulate the attack for the first four cookies was done on a server equipped with two 14-core Intel Xeon 2.6GHz CPUs, whereas the generation of ciphertext data for the remaining cookies was carried out using Amazon Web Services (AWS) computing infrastructure. In total, the generation of ciphertext data required the equivalent of approximately 73,000 core-hours of computation on AWS; for the AWS computations, we used the largest available compute-optimized instances denoted `c4.8xlarge`. The part of the attack corresponding to running the list Viterbi algorithm was done on a server equipped with two 8-core Intel Xeon 3.3GHz CPUs.

Results Figure 6 shows the measured success rate of the attack for different values of N. Each curve corresponds to an attack using the list Viterbi algorithm with a list size of either $L = 1$, $L = 2^{10}$, or $L = 2^{15}$. We note that even for $L = 2^{15}$ and $N = 8 \cdot 2^{42}$, two out of the 16 considered cookies were not recovered. Both of these cookies contain 6-bit substrings leading to some of the smallest values in the table for $\Pr[\texttt{Xb64}[\theta] = Y]$ (these values are `0x2c`, `0x2e`, and `0x39`; see Figure 2 for a visual representation of the biases for these values). The poorer

performance of our attack in these cases is consistent with our discussion in Section 5.3. We are confident that with additional ciphertexts, these cookies would be recovered as well.

Generally, the recovery pattern is similar to the TLS Scramble attack, albeit higher values of N are required; the attacks using $L = 2^{15}$ and $L = 2^{10}$ perform significantly better than the attack for $L = 1$, while the difference between the performances for two larger values for L is less significant. Note that since only 16 attacks were simulated, the results shown in Figure 2 represent a more noisy measurement of the attack performance than the corresponding measurements for TLS Scramble, which was based on 128 independent attack simulations.

7 Conclusions

We have presented a comprehensive analysis of the two cookie masking methods of [10] when used in conjunction with RC4. We used the method of maximum likelihood estimation to develop statistically rigorous attacks, coupled with extensive simulations to estimate their performance. In the case of TLS Scramble, we were able to adapt the techniques developed in [4] to perform a theoretical evaluation of the number of ciphertexts needed to guarantee that the attack is successful with high probability. In both cases, we make use of the list Viterbi algorithm to extend our basic attacks recovering adjacent pairs of cookie symbols to full attacks recovering complete cookies. Our analysis indicates that the cookie masking methods presented in [10] do enhance the security of RC4 in SSL/TLS when it is used to protect HTTP cookies, but not sufficiently to warrant deploying the mechanisms as they stand.

It is interesting that, while the Base64 encoding of the masked cookie in MCookies was presumably not intended as a security measure, the presence of this encoding step means that our attack against MCookies becomes significantly harder to mount in practice compared to our attack on TLS Scramble. More precisely, the cost of the attack rises from around 2^{37} ciphertexts to around 2^{45}.

For our MCookies analysis, it would be useful to develop a better understanding of the distributional properties of the simplified log-likelihood function given in equation (7), and use this to perform an extended analysis (possibly using order statistics) to make predictions about the number of ciphertexts needed to recover θ^* correctly. It would also be interesting to extend our analysis to handle masking mechanisms using multiple masks, as hinted at in [10] as being a possible extension of MCookies. In particular, it would be useful to evaluate how much extra security would be obtained by involving additional masks.

Acknowledgements

This research was supported by a generous donation of computing resources by Amazon Web Services.

Paterson was supported in part by a research programme funded by Huawei Technologies and delivered through the Institute for Cyber Security Innovation

at Royal Holloway, University of London. Schuldt was supported in part by JSPS KAKENHI Grant Number 15K16006.

References

1. Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the Security of RC4 in TLS. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 305–320, Berkeley, CA, USA, 2013. USENIX Association.
2. Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.
3. Adam Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.
4. Remi Bricout, Sean Murphy, Kenneth G. Paterson, and Thyla van der Merwe. Analysing and exploiting the Mantin biases in RC4. *IACR Cryptology ePrint Archive*, 2016:63, 2016.
5. Christina Garman, Kenneth G. Paterson, and Thyla van der Merwe. Attacks only get better: Password recovery attacks against RC4 in TLS. In Jung and Holz [9], pages 113–128.
6. Sourav Sen Gupta, Subhamoy Maitra, Goutam Paul, and Santanu Sarkar. (non-)random sequences from (non-)random permutations - analysis of RC4 stream cipher. *J. Cryptology*, 27(1):67–108, 2014.
7. Takanori Isobe, Toshihiro Ohigashi, Yuhei Watanabe, and Masakatu Morii. Full plaintext recovery attack on broadcast RC4. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 179–202. Springer, 2013.
8. Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.
9. Jaeyeon Jung and Thorsten Holz, editors. *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. USENIX Association, 2015.
10. Olivier Levillain, Baptiste Gourdin, and Hervé Debar. TLS Record Protocol: Security Analysis and Defense-in-depth Countermeasures for HTTPS. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 225–236. ACM, 2015.
11. Itsik Mantin. Predicting and distinguishing attacks on RC4 keystream generator. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2005.
12. Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: Exploiting the SSL 3.0 fallback, September 2014.
13. Toshihiro Ohigashi, Takanori Isobe, Yuhei Watanabe, and Masakatu Morii. How to recover any byte of plaintext on RC4. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 155–173. Springer, 2013.

14. Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. Big bias hunting in Amazonia: Large-scale computation and exploitation of RC4 biases (invited paper). In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 398–419. Springer, 2014.
15. Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. Plaintext recovery attacks against WPA/TKIP. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 325–349. Springer, 2014.
16. Kenneth G. Paterson and Mario Strefler. A practical attack against the use of RC4 in the HIVE hidden volume encryption system. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 475–482. ACM, 2015.
17. Andrei Popov. Prohibiting RC4 Ciphersuites. RFC 7465, February 2015.
18. Pratik Guha Sarkar and Shawn Fitzgerald. Attacks on SSL – a comprehensive study of BEAST, CRIME, TIME, BREACH, Lucky 13 and RC4 biases, August 2013.
19. Santanu Sarkar, Sourav Sen Gupta, Goutam Paul, and Subhamoy Maitra. Proving tls-attack related open biases of RC4. *Des. Codes Cryptography*, 77(1):231–253, 2015.
20. Nambi Seshadri and Carl-Erik W. Sundberg. List Viterbi decoding algorithms with applications. *IEEE Transactions on Communications*, 42(234):313–323, 1994.
21. Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In Jung and Holz [9], pages 97–112.