Non-uniformities in the RC4 Stream Cipher

Simon Campbell
under the supervision of
Prof. Andrew Ker

Dissertation submitted for the completion of the
MSc in Computer Science

University of Oxford

Trinity Term 2015

Abstract

The RC4 stream cipher is used to protect messages from eavesdroppers in many settings, including some of the Transport Layer Security (TLS) protocols used to secure much internet traffic. Non-uniformities in the output of a stream cipher are a weakness that an eavesdropper can exploit to gain information about the encrypted messages. In this report we verify the size of some non-uniformities of RC4 in TLS that were recently reported based on experimental observations by Al Fardan et al. (2013). We extend this work to report on several newly discovered non-uniformities. We also report methods and results quantifying the vulnerability to eavesdropping of messages encrypted by RC4 in TLS as a result of these non-uniformities. Based on the resulting analysis, and in conjunction of the work of others, we add our voice to those urging that RC4 no longer be used.

Contents

1 Introduction **3**

2 The Problem to be Studied **5**
2.1 Background 5
2.1.1 The RC4 Stream Cipher 5
2.1.2 What is a Non-Uniformity? 6
2.1.3 Predicted and Observed Non-Uniformities in RC4's Output 7
2.2 The Problem to be Studied 8
2.2.1 Single Byte Biases 9
2.2.2 Double Byte Biases 9

3 Methodology **11**
3.1 Identifying and Quantifying Non-Uniformities 12
3.1.1 Identifying Biases 12
3.1.2 Quantifying Biases 12
3.1.3 Data Requirements 13
3.2 Data Collection 14
3.2.1 Source of Pseudo Randomness 14
3.2.2 Code Speed and Computing Resources 15
3.3 Measuring the Loss of Randomness 22
3.3.1 Empirical Entropy Estimation 22
3.4 An Attacker's Perspective 26
3.4.1 The Attacker 26
3.4.2 Measuring RC4's Vulnerability to Our Attacker 28

4 Results **33**
4.1 Single Byte Biases 33
4.1.1 Biases in the First 256 Bytes 33
4.1.2 Biases in the Second 256 Bytes 37
4.2 Double Byte Biases 43
4.2.1 Fluhrer-McGrew Biases 43
4.3 Vulnerability Analysis 43

5 Conclusions **52**
5.1 Scientific Conclusions . . . 52
5.1.1 Confirmed and Unconfirmed Biases . . . 52
5.1.2 Discovered Biases . . . 52
5.1.3 Measuring Vulnerability . . . 53
5.1.4 Potential Future Extensions . . . 54
5.2 Other Insights Gained . . . 54
5.2.1 Coding for Speed . . . 54
5.2.2 Continued Use of RC4 . . . 54
5.2.3 Lessons for Crypto System Selection . . . 55

Chapter 1

Introduction

"*The nail that sticks out gets hammered down.*" - Japanese Proverb

Uniformity is useful when you want to pass unrecognised, it lets you blend into the crowd. In the field of cryptography, we aim to pass messages so that our adversaries will not be able to read them. If the way we encrypt our messages makes all messages appear indistinguishable then we are making things hard for our opponents. Conversely, features which distinguish the encryptions of particular messages can be exploited by our enemies, and they may be able to read the content we've tried to hide.

The **RC4 stream cipher** is an encryption system that has found wide applications since the early 1990's. For example, RC4 has been included in the Secure Socket Layer/Transport Layer Security (SSL/TLS) protocols[1] used to secure much internet traffic. Therefore, vulnerabilities in this system are worth investigating and it is of interest whether this particular system is providing enough uniformity to keep our messages unread by those who might want to uncover them. In this report we address this issue by seeking to answer two questions:

1. How uniform (or not) is the output of the RC4 encryption system? And, as a result,
2. How vulnerable is the system to an attacker who seeks the messages being sent?

In recent years there have been a series of papers published focussing on these questions, in part because RC4's use had increased in response to vulnerabilities in other TLS encryption systems (e.g. the BEAST attack [4]). We will seek to replicate some of the recently observed results regarding non-uniformities/biases, in particular the results found in Al Fardan et al [3]. We will also extend beyond the existing literature to see whether we can discover any previously unknown non-uniformities. We will then consider how vulnerable these non-uniformities make the system. As a preliminary we will lay out the structure of this report.

The Structure of this Report

This report is made up of 4 further chapters. In the second chapter we will lay out some of the relevant background to our work. This will include a description of the RC4 stream cipher and what

[1] The latest (draft) version of TLS 1.3 is at http://tools.ietf.org/html/draft-ietf-tls-tls13 and includes references to prior versions

we mean by a non-uniformity in its output. We then give an overview of the literature containing predictions and observations of non-uniformities/biases. We will then state the questions we will seek to address in this report in formal and precise terms.

In chapter 3 we will describe the methodology we will follow to answer these questions. This will provide a description of how we will go about identifying and quantifying the non-uniformities in the RC4 output. We will also outline how we went about collecting the data required for this analysis. Finally, we go on to report the methods to be used to quantify how vulnerable the RC4 cipher is to an attacker as a result of the observed non-uniformities.

In chapter 4 we report on the results obtained by following this methodology. This includes the details of several previously undiscovered non-uniformities.

In chapter 5 we seek to draw conclusions from our work. We do so both with regards to the specific scientific results uncovered regarding RC4, and to the wider implications of our work. Without further ado we will proceed to chapter 2.

Chapter 2

The Problem to be Studied

2.1 Background

2.1.1 The RC4 Stream Cipher

Let $\mathcal{B} = \{0, 1, \ldots, 255\}$ be the set of all possible single byte values. We will at various times find it convenient to refer to these values interchangeably with their canonical binary representations, $\{0,1\}^8 = \{00000000, 00000001, \ldots, 11111111\}$. The RC4 stream cipher is a map

$$\text{RC4}\colon \mathcal{K} \times \mathcal{M} \to \mathcal{C},$$

where $\mathcal{K} = \mathcal{B}^l$ is the space of secret keys, and l is typically between 5 and 32, $\mathcal{M} = \mathcal{B}^*$ is the message space and $\mathcal{C} = \mathcal{B}^*$ is the ciphertext space. For the implementation used in TLS, $l = 16$ and throughout this report, unless otherwise stated, we shall assume this value is taken.

RC4's operation is defined by two algorithms, the **Key Scheduling Algorithm** (KSA) and the **Pseudo Random Generation Algorithm** (PRGA) operating in conjunction with RC4's internal state. This state, st, consists of the triplet $(i, j, \mathcal{S})$ where $i, j \in \mathcal{B}$ and $\mathcal{S}$ is a byte array containing a permutation of all of the byte values in $\mathcal{B}$.

The definition of the KSA and PRGA are shown as Algorithms 1 and 2. If we take $r \geq 1$ and we let m_r be the rth byte of the message, and let Z_r be the rth byte output by the RC4 PRGA, then the rth byte of the ciphertext output by RC4, C_r, is defined by $C_r = m_r \oplus Z_r$, where here, and throughout, $\oplus$ is the bitwise XOR operation when the two values are written in their canonical binary form. We call the values $\{Z_r\}$ the **key-stream** and the values $\{C_r\}$ the **cipher-stream**. This completely defines to operation of the RC4 stream cipher.

The RC4 stream cipher was originally designed by Ron Rivest in 1987 and the details of its operation were proprietary to RSA Security. In 1994 the algorithm was successfully reverse engineered and published anonymously on the cypherpunks mailing list.[1] Since this publication the algorithm has found widespread use including the WEP and WPA wireless protocols [1, 2] as well

[1] The original post can be found at http://web.archive.org/web/20080404222417/http://cypherpunks.venona.com/date/1994/09/msg00304.html

Input: Key K, of length l bytes
Output: An initial internal state st_0
begin
  for $i = 0, \ldots, 255$ **do**
    $\mathcal{S}[i] \leftarrow i$;
  end
  $j \leftarrow 0$;
  for $i = 0, \ldots, 255$ **do**
    $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$;
    swap$(\mathcal{S}[i], \mathcal{S}[j])$;
  end
  $i, j \leftarrow 0$;
  $st_0 \leftarrow (i, j, \mathcal{S})$;
  return st_0;
end

Algorithm 1: The RC4 Key Scheduling Algorithm which is run once to initialise the internal state. All addition involving the state variables is mod 256.

Input: An internal state st_r
Output: The succeeding internal state st_{r+1} and key-stream output byte Z_{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 (st_{r+1}, Z_{r+1});
end

Algorithm 2: The RC4 Pseudo Random Generation Algorithm. All addition involving the state variables is mod 256.

as the SSL/TLS protocols and many other proprietary settings.

Over time weaknesses were uncovered in RC4 and the way it was being utilised. Perhaps the most penetrating example was the discovery of weaknesses in the KSA reported in [5] which lead to a catastrophic attack on RC4's implementation in WEP as described in [15]. This and other work did not lead to similar problems in the implementation of TLS due to several factors including a difference in the way RC4 is utilised. Unfortunately, other vulnerabilities have been uncovered that do apply in the TLS setting. These relate to non-uniformities in RC4's output.

2.1.2 What is a Non-Uniformity?

Any stream cipher encrypts messages by $\oplus$-ing the message's bytes with a pseudo-random stream of key-stream bytes. From an encryption standpoint, an ideal stream cipher would be a one-time-

pad with truly random and independent bytes used for the key-stream.[2] Suppose we have such a stream cipher. Let E be some event in the key-stream e.g. '$Z_2 = 0$', where Z_2 is the second byte of the key-stream. Then let $P_R[E]$ be the probability of event E taken over the truly random and independent generation of the key-stream bytes. It should be clear that $P_R[Z_2 = 0] = 1/256$. It is a result of Shannon's that using this cipher and given only the ciphertext of any message, an attacker gains no information about the encrypted message. We have a theoretically perfect cipher.

Now we can also consider the probability of the event E in the output of the RC4 PRGA. This probability can be taken over the random selection of the RC4 key, K.[3] If we define this probability by $P[E]$ then we say that there is a **non-uniformity** or **bias** in the output of RC4 if for any event E,

$$P[E] \neq P_R[E].$$

The value $P[E] - P_R[E]$ gives us the size and direction of the non-uniformity (towards or away from E).

Why are we interested in these non-uniformities?

On a theoretical level, the counterpart to Shannon's theorem regarding a perfect cipher is that any non-uniformity in a stream cipher provides an eavesdropper with the theoretical capability to gain information about the encrypted message from ciphertexts. Knowledge of non-uniformities provides an attacker with a better than random chance of guessing the stream cipher's output, Z_r. This can provide the attacker with enough information to learn about the value of m_r by observing the ciphertext C_r. In the worst case this leads to potential Ciphertext Only Attacks (COAs). In practice a non-uniformity could be so small or require such a long message that an attacker would never practically be able to gain useful information. Unfortunately, as we shall see, RC4 has non-uniformities that do not escape practical exploitation.

2.1.3 Predicted and Observed Non-Uniformities in RC4's Output

In the last 14 years a range of non-uniformities in the output of RC4 have been identified. In 2001 Fluhrer and McGrew [6] predicted long-term biases in the RC4 key-stream for various value of byte pairs in neighbouring positions of the stream. Mantin and Shamir [10] also reported a significant bias towards $Z_2 = 0$. At the time they predicted that there were no other biases towards $Z_r = 0$ for $3 \leq r \leq 256$. However, in 2011 Maitra et al. reported theoretical reasoning to suggest that there would be biases towards 0 at all of these positions. These and other similar single byte position biases will be an area of focus in our work.[4]

[2]Practically, this suffers from the limitation of requiring that we secretly share the full pad between Alice and Bob ahead of time.

[3]In the TLS setting, the selection of K can effectively be treated as random.

[4]These and the later predictions of biases were derived in similar ways. They involve finding an event I in the internal state of RC4 such that the key-stream event E obeys the rule $P[E \mid I] \neq P_R[E]$ and $P[E \mid I^{\complement}] = P_R[E]$. In general this gives $P[E] = P[E \mid I^{\complement}] \cdot P[I^{\complement}] + P[E \mid I] \cdot P[I] \neq P_R[E]$. The reasoning used in the literature is generally not conclusive as the condition $P[E \mid I^{\complement}] = P_R[E]$, whilst often reasonable, is usually assumed in the literature but not proved. Experimental validation of the biases is therefore important.

In the meantime Mantin reported in [9] on the existence of biases towards sections of RC4 output of the form $Z_r, Z_{r+1}, \ldots, Z_{r+g+2}, Z_{r+g+3} = A, B, S, A, B$ where A and B are byte values and S is a string of g byte values. This and similar biases will not be a focus for us here but they are not insignificant.

There were other biases reported prior to 2011 and [14] contains a comprehensive list of the literature. Subsequent to 2011 there has been a flurry of discoveries reported. Sen Gupta et al. in [14] reported further theoretical justifications for the biases towards $Z_r = 0$ for $3 \leq r \leq 256$ and furthermore provided predictions of additional biases in the distribution of Z_1, key length dependent biases towards $Z_l = 256 - l$ and many other biases in the internal state of RC4.

In 2013 Al Fardan et al. [3] and Isobe et al. [8] contemporaneously sought to experimentally verify and discover biases in the first 256 bytes of RC4 key-stream output and, using these and other observed biases, sought to convert this knowledge into specific attacks, including COAs revealing cookies in TLS sessions. These attacks take place in the **Broadcast** setting where an attacker can view multiple encryptions of the same message using randomly selected keys. This setting is possible when Alice is using TLS through a web browser. An attacker may be able to trigger/force re-keying of the RC4 cipher followed by the resending of the same message encrypted under the new (pseudo random) key, as demonstrated in [3].

Isobe et al. [8] and Sarkar et al. [12] sought to provide theoretical justifications for the existence of the newly discovered biases in the first 256 byte positions. Ohigashi et al. [11] extended the attacks in [8] and the attacks presented in [3] were further developed in early 2015 in Garman et al. [7] with the results moving much closer to practical exploitation. This was the state of play when we began our work.

These developments contributed to a recent proposal in February 2015 to entirely remove RC4 from the standard TLS cipher suite.[5] Despite this suggestion, RC4 continues to be used to encrypt internet traffic with a recent survey by the ICSI Certificate Notary showing c. 21% of TLS traffic using this cipher.[6] (Although, encouragingly, this is down from the c. 50% figure reported in [3] in 2013). Exploring non-uniformities in RC4 is therefore still relevant in the current environment and will hopefully help inform the design and selection of ciphers in future as well.

2.2 The Problem to be Studied

In our work we will seek to verify the non-uniformities observed by Al Fardan et al. [3], namely the biases in the distributions of the first 256 key-stream bytes and the Fluhrer-McGrew double byte biases. We will also extend these results by observing the distributions of the key-stream byte values Z_r for $257 \leq r \leq 512$ as well. This will reveal previously unreported biases. We will then consider the vulnerabilities of the RC4 stream cipher that result from some of these biases, namely the non-uniformities in the distributions of the first 512 key-stream byte values.
The precise biases that we will seek to verify from [3] are detailed below but first we specify some notation.

[5] http://tools.ietf.org/html/rfc7465
[6] http://notary.icsi.berkeley.edu/

A note on terminology When discussing RC4 key-stream output we shall reference byte positions r, where $r \geq 1$. Each byte position has a key-stream byte output value, denoted by Z_r, where $Z_r \in \mathcal{B}$. It will be common when discussing the distribution of values for Z_r over the random selection of the RC4 keys to consider what we call the **second order magnitude biases**, $\boldsymbol{b_a^r}$ defined by the distribution of Z_r as follows, $\forall r \geq 1$, $a \in \mathcal{B}$,

$$P[Z_r = a] = \frac{1}{256} + \frac{b_a^r}{256^2}$$

These values provide a useful scale to measure non-uniformities in the single byte distributions.

For the double byte biases we are considering long-term biases that are conditional on the value of the internal state variable i at the time that Z_r is output (note that at this moment $i = r \bmod 256$). For convenience we define the **conditional biases of magnitude order 1.5**, $\boldsymbol{\overline{b}_{a,b}^i}$ as follows $\forall r > 1024$, and i, a, $b \in \mathcal{B}$,

$$P[Z_r = a \wedge Z_{r+1} = b \mid r = i \bmod 256] = \frac{1}{256^2} + \frac{\overline{b}_{a,b}^i}{(256^2)^{1.5}}$$

From time to time we will also refer to the portion of key-stream output defined by $\{Z_r : 256(k-1) < r \leq 256k\}$ for some $k \geq 1$ as the $\boldsymbol{k}^{\mathbf{th}}$ **page** of key-stream output. $r = 256k$ is a natural page break as this is when the value of i cycles back to 1.

We now turn to the specific non-uniformities reported or identified in [3] and in part drawn from [14, 10, 6].

2.2.1 Single Byte Biases

We will seek to verify the existence and size of the following single byte biases:

- $P[Z_2 = 0] \approx \frac{1}{128}$ i.e. $b_0^2 \approx 256$
- $b_0^3 = 0.351089$ and $b_0^4, b_0^5, \ldots, b_0^{255}$ is a decreasing sequence with terms that are bounded as follows
$$0.242811 \leq b_0^r \leq 1.337057$$
- For keys of length l bytes, $b_{256-l}^l \geq 1$.
- For keys of length 16 bytes (as used in TLS) $b_{129}^1 < 0$.
- For positions $1 \leq r \leq 256$ in the key-stream output, $b_{r \bmod 256}^r > 0$ and, for r that is a multiple of (the key length) 16, $b_{256-r}^r > 0$.

2.2.2 Double Byte Biases

We also seek to verify the existence and size of the double-byte biases shown in table 2.1. They relate to immediately sequential pairs of bytes in the key-stream output beyond just the initial 256 bytes (in fact Al Fardan et al. and we ignore the initial 1024 bytes).

Value(s) of $(a, b) = (Z_r, Z_{r+1})$	Value(s) of $i = r \bmod 256$	Approximate value of $\overline{b}^i_{a,b}$
$(0, 0)$	$i = 1$	$+2$
$(0, 0)$	$i \neq 1, 255$	$+1$
$(0, 1)$	$i \neq 0, 1$	$+1$
$(i + 1, 255)$	$i \neq 254$	$+1$
$(255, i + 1)$	$i \neq 254$	$+1$
$(255, i + 2)$	$i \neq 253$	$+1$
$(129, 129)$	$i = 2$	$+1$
$(255, 255)$	$i \neq 254$	-1
$(0, i + 1)$	$i \neq 0, 255$	-1

Table 2.1: The Fluhrer-McGrew double byte biases first predicted in [6].

As previously mentioned we will also seek to quantify the vulnerability of the RC4 stream cipher as a result of the single byte biases we observe. The precise methodology for this will be detailed in the next chapter.

Chapter 3

Methodology

In this chapter we describe the methodology we will follow. In the first section we describe how we will go about identifying and quantifying non-uniformities in RC4's key-stream output. This will involve statistical hypothesis testing and generating confidence intervals for the biases we observe. The second section of this chapter describes how we went about collecting the data required for this initial analysis. This includes details of the effort we went to in optimising our code to allow us to collect this data. These first two sections describe the work that will verify and extend the observations of biases published in [3].

In the remaining two sections we consider methods for measuring the vulnerability of the RC4 cipher as a result of the observed non-uniformities. In section three we shall consider the feasibility of using entropy as a metric for this purpose. Entropy is in theory the obvious method for measuring randomness/uniformity and so is worthy of consideration. However, we shall perform a mathematical derivation which implies that measuring entropy is not practicable at this time for our purposes. In section 4 of this chapter we consider alternative methods by viewing the question of vulnerability from the perspective of an attacker and developing metrics based on this.

The resulting quantification will depend on the specific attack used, so we will describe the attacks considered. Here we only consider attacks on the single byte biases as these appear to be the most practicable. The single byte non-uniformities depend on the byte number (r, the position in the output stream) so the measurements will consider each position separately. The method of quantification could take several forms, two that will be considered are:

- Measuring the average number of repeated encodings of any given message that are required to get the average number of guesses required to identify the correct message byte below a fixed threshold, and,
- Measuring the average number of repeated encodings of any given message that are required to get a fixed level of certainty as to the message byte value on the first guess.

These will be justified by considering two different attack settings, one where the attacker has multiple chances to guess the correct message byte values and the other where she has only one.

3.1 Identifying and Quantifying Non-Uniformities

We seek to observe biases in the distribution of outputs of the RC4 key-stream. The distributions we seek to measure are over the random selection of the RC4 key (which determines all subsequent output from the stream). We will therefore collect output from some number of (pseudo) randomly generated keys and examine the resulting distribution of the outputs for non-uniformities. We will identify and confirm non-uniformities by performing statistical hypothesis tests for the existence of the previously reported biases. This will simply be to test whether there is a bias from the uniform distribution. We will then quantify the biases by deriving confidence intervals for the relevant probabilities.

All of the statistics we observe in this part of the investigation are binomially distributed. We are simply counting the number of occurrences of a particular cipher stream event which has a fixed probability. However, in all cases considered, the number of observations (N) and the probability of the event being observed (p) are such that the normal distribution approximation is accurate. For simplicity's sake we therefore employ this approximation when testing hypotheses and deriving confidence intervals in the subsequent analysis.

3.1.1 Identifying Biases

We here describe the method followed for testing our hypotheses. Let E be the event under consideration and $p = P[E]$ be the probability that we seek to quantify. Let N be the number of independent observations of the event under consideration (e.g. in the single byte case, N is the number of pseudo randomly selected keys used). In each hypothesis test we define p_{H_0} to be the value assumed by p under the Null Hypothesis. We define $\sigma_{H_0} = \sqrt{p_{H_0}(1 - p_{H_0})/N}$, the standard deviation of the normal distribution approximation to $Bin(N, p_{H_0})/N$. So we will calculate hypothesis test values (Z scores) using the normal distribution $N(p_{H_0}, \sigma_{H_0})$. So, if the observed experimental probability is,

$$\hat{p} = \frac{\text{No. of occurences of } E}{N}.$$

Then the observed Z score value is

$$\frac{\hat{p} - p_{H_0}}{\sigma_{H_0}} = \frac{\hat{p} - p_{H_0}}{\sqrt{p_{H_0} \cdot (1 - p_{H_0})/N)}}.$$

We employ a 99.99% confidence level for all tests as a means to be very confident of our results. Therefore a two sided test will have threshold Z scores of ± 3.891.

3.1.2 Quantifying Biases

We here describe the method followed for deriving confidence intervals for the probabilities examined (i.e. $P[E]$). Having observed $\hat{p}$ we can estimate the standard deviation for the distribution $Bin(N, p)/N$ as $\hat{\sigma} = \sqrt{\hat{p}(1 - \hat{p})/N}$. Using the normal distribution approximation for the distribution of $\hat{p}$, $N(\hat{p}, \hat{\sigma})$, we can derive a 99.99% confidence interval for p as follows

$$(\hat{p} - 3.891\hat{\sigma}, \hat{p} + 3.891\hat{\sigma}).$$

We use 99.99% intervals because we want to be very sure of the estimates we give. If this were to require more data than was feasible we would have considered reducing the intervals to the more

traditional 95% which would roughly halve the width of the intervals. In the end this was not necessary. This can also be converted to a confidence interval for the relevant bias term by simple linear scaling e.g. if E is the event $Z_r = a$, by definition of the second order magnitude bias for single bytes the relevant confidence interval for b_a^r is

$$\left(256^2 \cdot \left[\hat{p} - 3.891\hat{\sigma} - \frac{1}{256}\right] \, , 256^2 \cdot \left[\hat{p} + 3.891\hat{\sigma} - \frac{1}{256}\right]\right).$$

Similarly, if E was the event "$Z_r = a$ and $Z_{r+1} = b$ given that $r = i \bmod 256$", then the relevant confidence interval for $\overline{b}_{a,b}^i$ is

$$\left(256^3 \cdot \left[\hat{p} - 3.891\hat{\sigma} - \frac{1}{256^2}\right] \, , 256^3 \cdot \left[\hat{p} + 3.891\hat{\sigma} - \frac{1}{256^2}\right]\right).$$

3.1.3 Data Requirements

We seek to quantify the biases as accurately as possible. However, we must consider how much data would be sufficient for our needs. The second order magnitude single byte biases reported in the theoretical literature are distinguishable in the first decimal place so if we can get confidence intervals to quantify the biases to 1 d.p we shall be providing useful data. If we can get the range of the interval to be less than 10^{-2} then this will be more than sufficient. This would require the range of the confidence interval to be limited as follows

$$2 \times 3.891\hat{\sigma} \times 256^2 \leq 10^{-2}$$

so,

$$2 \times 3.891 \times \sqrt{\frac{\hat{p}(1-\hat{p})}{N}} \times 256^2 \leq 10^{-2}$$

so, using the rough approximation of $\hat{p} \approx 1/256$, we require

$$N \geq \left[2 \times 3.891 \times \sqrt{\frac{1}{256}(1 - \frac{1}{256})} \times 256^2 \times 10^2\right]^2 \approx 2^{43.20}.$$

[3] sampled 2^{44} random keys in order to observe the single byte biases for exploitation.

When it comes to collecting data to measure the double byte biases we assume that for any key, in the long term, successive pages of key-stream output are independent. In particular, we assume this holds after output byte position 1024. (This assumption is implicit in [3]). The conditional biases of magnitude order 1.5 predicted in the literature are of size 1 so if we can limit the range of the confidence interval to 10^{-1} that would be useful. By a similar calculation to that carried out above for the single byte biases this gives a lower bound on N of 2^{53}. In fact, due to computing resource constraints we were only able to collect $N = 2^{45}$ observations. We shall see that this is sufficient to confirm the biases, although greater accuracy in quantifying them would have been nice to have. We note that [3] collected $N = 2^{42}$ observations.

3.2 Data Collection

We now describe how we went about collecting the data required for our analysis. To collect the required data we wrote code in C++ (2011 standard). The pseudo-code shown as algorithm 3 details the general procedure that we implemented.

```
Input: k the number of keys required, and L the number of bytes of RC4 output required
       per key
Output: hist, a histogram, indexed by events E which counts the number of observations of
        each event, N the total number of observations made.
begin
  for i = 0, ..., k − 1 do
    Generate a (pseudo) random key, K;
    rc4Stream.keySchedule(K); // Key Schedule an RC4 stream
    for r = 1, ..., L do
      Z_r ← rc4Stream.PRGA(); // Collect the next key-stream byte from the RC4
      Stream
      if an outcome event E can be parsed from the results (Z_1, ..., Z_r) then
        N ← N + 1;
        hist[E] ← hist[E] + 1;
      end
    end
  end
  return hist, N;
end
```

Algorithm 3: Our general data collection procedure.

This suggests two immediate questions. What source of pseudo randomness will be used? And, will our code be fast enough to provide sufficient data with the available computing resources? We address these questions in turn.

3.2.1 Source of Pseudo Randomness

It is something of an irony that in order to investigate how random RC4 is we need a source of randomness that is sufficiently random. This is to ensure that the biases we observe are from RC4 and not from the source of our keys. Previous work in this area, including [3] and [16], has used AES in counter mode as a source of pseudo random data for similar purposes as we are pursuing. This has the benefit of being believed to be a good pseudo random number generator which has support in hardware instruction sets (allowing for faster running times in theory). It has the downside that the period for the output stream is $\leq 2^{128}$ which requires some care when collecting large data sets. We decided to utilise the Mersenne Twister pseudo-random number generator that is part of the standard library for C++. This has the benefit of running sufficiently quickly whilst not requiring hardware level instruction implementation. The algorithm also has a much larger period of $2^{19937} - 1$ and is known to pass many tests for statistical randomness. The much larger state space also helps ensure that, using different seeds for parallel runs, we are much less likely to produce duplicate data from parallel runs (we ultimately used pseudo random seeds for each parallel run to help keep this probability low). Furthermore, since, in part, we are seeking to check

the observation of known biases as reported in earlier work (particularly in [3]), it is helpful to utilise a different source of pseudo randomness to that earlier work to eliminate the possibility that any observations were artefacts of the generator selected.[1]

3.2.2 Code Speed and Computing Resources

The data collection phase is anticipated to require the collection of output from a significant number of RC4 output streams keyed by randomly selected keys. The available computing resources include 8×16 cores of Intel E5-2640v5 Haswell architecture made available through the University of Oxford Advanced Research Computing facility. The clock speed of these cores is 2.6 GHz. We were limited in the amount of time that we could use these cores for as they are a shared resource. We could only run code for a maximum of 5 days at a time. Furthermore, our total time on the machines was limited to c.10 days. Other resources are available for use, but they consist of orders of magnitude fewer cores and so ideally we wished to complete all of the main data collection on the shared resources. Once we had learnt to use the ARC systems we ultimately utilised a total of c.1042 CPU days.

To ensure that the code will be able to run sufficiently quickly we performed some timing tests on our code. These tests were first performed on versions of the single byte collection code to identify any problematic bottlenecks and optimise as necessary. The same underlying class code was used to collect the double byte data with further optimisation performed as described below.

Single Byte Code Optimisation

As earlier discussed, for the single byte data collection, we would like to collect data for at least $2^{43.20}$ keys. In running our code we will want to periodically dump the data collected to ensure an external failure does not cause us to lose all of the data collected so far. Therefore, we actually set our lower bound number of keys at $2^{43.21}$ as this gives us a whole number of loops in the program when we divide the work as follows, 74 batches $\times$ 2^{30} keys per batch per core $\times$ 2^7 cores $\approx 2^{43.21}$ keys in total. Each of the 74 batches can finish with a data dump.

To collect the single byte experimental probabilities we ran the code shown in code snippet 3.1 as our implementation of the data collection procedure (algorithm 3).

To generate the required data will require that this (or equivalent) code run sufficiently quickly. If we attempt $2^{43.21}$ pages and say t is the time in seconds to run 2^{20} pages (i.e. c. 1,000,000 pages) then the total time to collect the targeted number of pages is:

$$\text{Total Time} = \frac{2^{43.21} \cdot t}{2^{20} \times 3600 \times 24} \text{days} \approx 2^{6.81} \cdot t \text{ days}$$

Since we have $8 \times 16 = 2^7$ cores at our disposal for 5 days at a time we would like to achieve a run time such that $2^{6.81} t / 2^7 \leq 5$ days which means $t \leq 5.7$ will be sufficient. We should also have in mind a baseline which is the best we might hope for. There are documented reports of RC4 code producing 1 byte of output every 7 CPU clock cycles [13]. On a 2.6 GHz cpu this would give a run time of

$$\frac{2^{43.21} \times 256 \text{bytes per page} \times 7 \text{cycles per clock}}{2.6 \times 10^9 \times 3600 \times 24} \approx 81 \text{ days}$$

[1] If we found any biases that were artefacts resulting from this use of AES then that would likely be a much more significant result in practical cryptography than anything else we discover regarding RC4.

Code Snippet 3.1: This snippet shows the single byte data collection loop implementing Algorithm 3

```
//loop to generate multiple rc4 stream outputs
for (int i = 0; i < loopcount; i++) {

  //random key generation
  randomSource.selectRandomKey(key);

  //rekey with the newly selected key
  rc4Stream.keySchedule(key);

  //run RC4 stream for the required number of bytes algorithm and collect the
      number of occurrences of each byte in each position in histogram counters
  for (int i = 0; i < STREAM_OUTPUT_LENGTH; i++) {
     histograms[i][rc4Stream.PRGRound()]++; //increment the relevant histogram count
  }
}
```

This is equivalent to $t \approx 0.7$ so our target of 5.7 is possible. We therefore begin by checking the in silica run time of the code and establishing whether improvement is required and if so where best to focus the optimisation efforts.

Initial Code Speed Observations

The initial test results produced are shown in Figure 3.1. Firstly this control test revealed a value of $t \approx 8.7$ which was too large for our purposes. Therefore, we needed to consider means of increasing the speed of the in silica run time (decreasing t) if we wished to collect this data.

The other tests involved removing sections of the code to see which functionality was contributing the most to the cumulative running time. The main bottlenecks at this stage were in the RC4Stream PRGRound and KeySchedule methods as when these methods were avoided the running time dropped most substantially.

Further investigation of the Key Scheduling method revealed that the main bottleneck was the permutation array scheduling loop (as opposed to the permutation array initialisation loop). In particular the call to get the next byte of the key for the key scheduling process.

As an initial attempt to improve performance on calls to access the key array and the permutation array we re-factored the class code to use statically allocated arrays (on the stack instead of the heap). And performed all of the object creation statically as well. By locating these objects on the stack we hoped to maintain locality in memory and improve cache hits, with a hopeful improvement in timing. This was partially successful.

To achieve further improvements we experimented with changes in the implementation of the PRGRound method and the keyScheduling algorithm with varying optimisation settings on compilation. The optimal timing that we discovered involved the code detailed in the following sections with the -Os optimisation flag on the Intel compiler. Interestingly, manually unrolling all loops and attempting to reorder code to take advantage of pipe-lining was not as beneficial as making the more minor changes shown here and allowing the compiler to handle further optimisation.

Test Name	Time Spent Initializing and Generating RC4 Streams (s)
Control Test	8.74436
Static Key Test	8.30614
No Re-Keying Test	5.22184
No RC4 Stream Gen. Test	4.53357
No Histogram Lookup Test	7.78659

Figure 3.1: The initial timing test results. The first row is the timings for the loop described in snippet 3.1 and in subsequent rows we have the results when adjustments to this loop are made by removing specific functionality/operations. All tests involved generating 2^20 streams of length 257.

PRGRound method

The initial (pre-optimisation) code is shown in Code Snippet 3.2. For the code after iterated

Code Snippet 3.2: This snippet shows the initial implementation of the RC4 PRGA (Algorithm 2). PERMUTATION_ARRAY_LENGTH has value 256 and is used for modular arithmetic.

```
virtual uint8_t PRGRound()
{
    _i = (_i + 1) % PERMUTATION_ARRAY_LENGTH;
    _j = (_j + _permutationArray[_i]) % PERMUTATION_ARRAY_LENGTH;

    //swap ith and jth elements
    uint8_t tmp = _permutationArray[_i];
    _permutationArray[_i] = _permutationArray[_j];
    _permutationArray[_j] = tmp;

    //return the output
    return _permutationArray[(_permutationArray[_i] + _permutationArray[_j]) %
        PERMUTATION_ARRAY_LENGTH];
}
```

optimisation the best results came when the only change was to use bit wise masking instead of the % operation for modular arithmetic, i.e. to replace "% PERMUTATION_ARRAY_LENGTH" with "& MASK_8".

keySchedule method

The is the initial code before optimisation is shown in Code Snippet 3.3. After our iterative optimisation efforts the main change related to removing modular arithmetic using the % operation and instead using bitwise masking. We also changed the RC4 Key class to remove use of %. This was done by copying the 16 byte key 16 times into an array that is $16 \times 16 = 256$ bytes long so that for $0 \leq i < 256$, the look up _key[i] gives the $(i \bmod 16)^{\text{th}}$ byte of the RC4 key without needing modular arithmetic.

Code Snippet 3.3: This snippet shows the initial implementation of the RC4 KSA (Algorithm 1).

```
virtual void keySchedule(RC4Stream::Key &key){

        //initialize the permutation array to be the identity permutation
        for (int i = 0; i < PERMUTATION_ARRAY_LENGTH ; i++) {
           _permutationArray[i] = i;
        }

        //schedule the permutation array
        unsigned int j = 0;
        for (int i = 0; i < PERMUTATION_ARRAY_LENGTH; i++) {
           j = (j + (unsigned int) _permutationArray[i] + (unsigned int)
               key.getModuloLength(i)) % PERMUTATION_ARRAY_LENGTH;
           //swap ith and jth elements
           uint8_t tmp = _permutationArray[i];
           _permutationArray[i] = _permutationArray[j];
           _permutationArray[j] = tmp;
        }
        //initialize the state variables
        _i = 0;
        _j = 0;
     }
```

setModuloLength method

The change in the implementation of the RC4Stream::Key class to store multiple copies of the key meant changing the method used to set the value of the key bytes. This method is used 16 times, every time a new key is generated, and so we also sought to maintain speed in this code. Pre-optimisation code is shown in Code Snippet 3.4. After optimisation, as mentioned earlier, the

Code Snippet 3.4: This snippet shows the initial implementation of the setModuloLength method. _key is a byte array of length KEY_LENGTH.

```
virtual void setModuloLength( int i, uint8_t byte ) {
             i = i % KEY_LENGTH;
             i = (i > -1 ? i : i + KEY_LENGTH);
             _key[i] = byte;
          }
```

internal storage of the key was extended to ensure that the key value could be repeated many times instead of requiring the modular lookup of the index sought, therefore _key is now of length PERMUTATION_ARRAY_LENGTH = 256. Instead of using an explicit loop we provided a pre-unrolled loop that attempted to take advantage of pipe-lining for enhanced performance. This code is shown in Code Snippet 3.5. In this case, this tactic helped performance.

These were the main changes made to the code after experimenting with a range of possible optimisations.

Code Snippet 3.5: This snippet shows the optimised implementation of the setModuloLength method.

```
virtual void setModuloLength( int i, uint8_t byte ) {
                i = i & MASK_4; //0 <= i < 16
     int j = i + 16;
     int k = i + 32;
                _key[i] = byte;
     _key[j] = byte;
     i += 48; // 48 <= i < 62
     _key[k] = byte;
     j += 48;
     _key[i] = byte;
     k += 48;following
     _key[j] = byte;
     i += 48; // 96 <= i < 112
     _key[k] = byte;
     j += 48;
     _key[i] = byte;
     k += 48;
     _key[j] = byte;
     i += 48; // 144 <= i < 160
     _key[k] = byte;
     j += 48;
     _key[i] = byte;
     k += 48;
     _key[j] = byte;
     i += 48; // 192 <= i < 208
     _key[k] = byte;
     j += 48;
     _key[i] = byte;
     k += 48;
     _key[j] = byte;
     i += 48; // 240 <= i < 256
     _key[k] = byte;
     _key[i] = byte;

      }
```

Optimised Results

The result of these optimisation efforts were that we managed to reduce t to c. 2.23.

Our initial aim was to get $t \leq 5.7$. We have therefore exceeded our aims. Handily, we have gotten the code to a point where our t value is less than half the necessary value. This implies that we may in fact be able to double the amount of data we collect as compared to our base requirement.

Given this fact we considered two options with respect to the Single Byte bias data. We could collect $N = 2^{44.21}$ observations for the first page of RC4 output instead of $2^{43.21}$ observations. Alternatively, we could still 'only' collect $N = 2^{43.21}$ observations but we could observe the first two pages of RC4 output instead of just the first page.

At the time of conducting the experiments, there was scant published data on the existence of biases in the second page of RC4 output. Given that the originally targeted observation count of $2^{43.21}$ was sufficient for our investigatory purposes in regards to the first page biases, we elected to extend our investigation to the second page as well, instead of increasing the observation count. This proved fruitful.

Double Byte Code Optimisation

For the double byte biases we need to count occurrences of events of the form "$Z_r = a$ and $Z_{r+1} = b$ given that $r = i \bmod 256$" where $0 \leq i, a, b < 256$. Naively, this would suggest using a histogram with dimensions $256 \times 256 \times 256$ to count the occurrences. Given the size of the data set required and because of the parallel computing we can get by with 32 bit unsigned integers to count any given histogram entry. This would imply a histogram for each core of size 2^{26} bytes (64 MB). Because subsequent lookups are unlikely to be spatially local in memory (the average distance would be c. 64 KB) this approach is likely to face speed barriers due to memory lookups/cache misses, and this was indeed the case. We trialled a range of histogram sizes by varying the first dimension size (collecting sub sets of the full space of events). The results of the tests can be seen in figure 3.2.

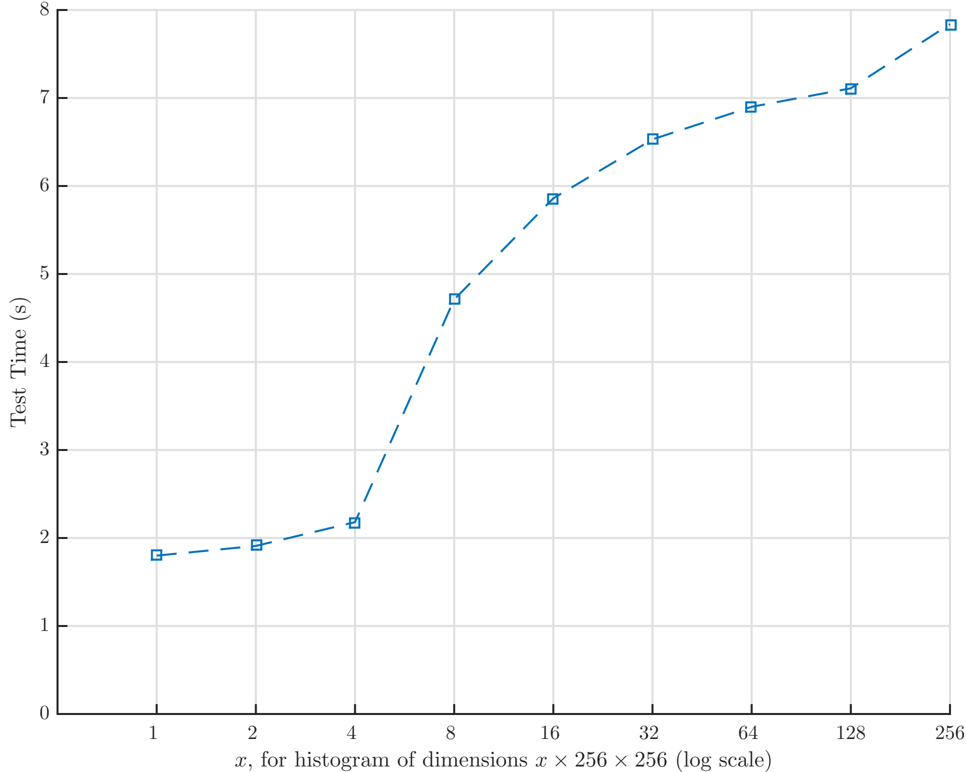


Figure 3.2: The observed timing to collect 2^{20} pages of double byte output data using histograms with varying sizes of initial index, from $1 \times 256 \times 256$ upto $256 \times 256 \times 256$.

The jump after size 4 is indicative of increased cache misses causing delays. The largest histogram that was workable given the computing and time constraints was a $4 \times 256 \times 256$ histogram[2]. This histogram size is large enough to collect data on all of the Fluhrer-McGrew double byte biases by sensibly allocating the space to collect the counts of selected events. There is a trade-off with speed in doing this, as any complicated lookup rules would slow down our code. The scheme we selected managed to provide data on all of the Fluhrer-McGrew events without material damage to the speed of our code. Under our scheme the histogram captured occurrences of the following events for $r > 1024$ and $a, b, i \in \mathcal{B}$:

$$r = i \bmod 256,\ Z_r = 0,\ Z_{r+1} = b$$
$$r = i \bmod 256,\ Z_r = 255,\ Z_{r+1} = b$$
$$r = i \bmod 256,\ Z_r = a,\ Z_{r+1} = 255$$
$$r = i \bmod 256,\ Z_r = 129,\ Z_{r+1} = b$$

These cover all of the Fluhrer-McGrew bias events.

This concludes the recounting of our code optimisation efforts.

[2]The implementation of this histogram was in fact 2MB in size, instead of 1MB. This was because we used unsigned 64 bit integers for collecting data, we ultimately decided to collate the counts for some events which resulted in a higher upper bound requirement for the data types.

3.3 Measuring the Loss of Randomness

We have already described the methods to be used in identifying the existence of the predicted single and double byte biases. We have also described the methods to be used in quantifying the individual biases. We now come to the final aspect of our analysis, quantifying the vulnerabilities of the RC4 cipher resulting from these non-uniformities. Simply put we wish to attempt to quantify how easily an attacker might exploit the available biases to carry out an attack and recover plaintext.

3.3.1 Empirical Entropy Estimation

One approach would be to attempt to estimate the entropy of the uni-gram (single byte) RC4 key-stream cipher output (over the uniform random selection of 16 Byte keys). Entropy is a theoretically justified metric of randomness (as unpredictability). A natural way to approach this would be to try and measure the entropy of the distribution of Z_r for each r. We could attempt this by calculating the plug-in entropy. If we let $\hat{p}_a$ be the empirical estimate of $p_a = P[Z_r = a]$ for some r, then the plug-in entropy for position r can be calculated as follows:

$$\hat{H} = -\sum_{a \in \mathcal{B}} \hat{p}_a \log_2(\hat{p}_a).$$

The maximal value obtainable is 8 bits and the scale of any deficit could be taken to indicate the level of weakness of the cipher at the given byte position.[3] Unfortunately, it turns out that the data we have available to us is insufficient to provide a useful estimate for the entropy. That is to say, our estimates for the entropy will be statistically indistinguishable from a uniform distribution's entropy. This is because, given the number of observations we have made of each byte position, the variance of the estimated entropy will be too large to provide a sufficiently small confidence interval to eliminate maximal entropy in most cases. To see why we shall first derive an expression for the variance and expectation.

Expectation and Variance of Experimental Entropy

We start by noting that

$$\hat{H} = -\sum_{a \in \mathcal{B}} \hat{p}_a \log_2(\hat{p}_a) = -\sum_{a \in \mathcal{B}} \hat{p}_a \frac{\ln(\hat{p}_a)}{\ln 2} \qquad \text{by definition.}$$

Thus, by linearity of expectation,

$$\mathbb{E}[\hat{H}] = -\frac{1}{\ln 2} \sum_{a \in \mathcal{B}} \mathbb{E}\left[\hat{p}_a \ln(\hat{p}_a)\right]. \tag{3.1}$$

Furthermore, since for different values of a, the $\hat{p}_a$ are only very weakly dependent, we can approximate the variance as follows

$$\mathrm{Var}[\hat{H}] \approx \frac{1}{(\ln 2)^2} \sum_{a \in \mathcal{B}} \mathrm{Var}[p_a \ln(p_a)] = \frac{1}{(\ln 2)^2} \sum_{a \in \mathcal{B}} \mathbb{E}[p_a^2 \ln(p_a)^2] - \mathbb{E}[p_a \ln(p_a)]^2. \tag{3.2}$$

[3] We shall see that this is a biased estimator for the entropy. There are other estimators that attempt to remove this bias but we shall see that this does not interfere with our argument.

Now we can consider the terms of this expression one by one. We will begin with $\mathbb{E}\left[\hat{p}_a \ln(\hat{p}_a)\right]$. Take some particular $0 \leq a < 256$. Since N, the number of observations, is sufficiently large relative to p_a we can estimate the distribution of $\hat{p}_a$ by

$$\hat{p}_a \sim N\left(p_a, \sigma_a^2\right),$$

$$\text{where } \sigma_a = \sqrt{\frac{p_a(1-p_a)}{N}}.$$

Therefore, we can say $\hat{p_a} = p_a + \varepsilon_a$ where

$$\varepsilon_a \sim N\left(0, \sigma_a^2\right)$$

is normally distributed noise. Now we can advance as follows:

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a \ln(\hat{p}_a)\right] &\approx \mathbb{E}\left[(p_a + \varepsilon_a)\ln(p_a + \varepsilon_a)\right] \\
&= \mathbb{E}\left[(p_a + \varepsilon_a)\left(\ln(p_a) + \ln\left(1 + \frac{\varepsilon_a}{p_a}\right)\right)\right] \\
&= \mathbb{E}\left[p_a \ln(p_a) + \varepsilon_a \ln(p_a) + (p_a + \varepsilon_a)\ln\left(1 + \frac{\varepsilon_a}{p_a}\right)\right] \\
&= \mathbb{E}\left[p_a \ln(p_a)\right] + \mathbb{E}\left[\varepsilon_a \ln(p_a)\right] + \mathbb{E}\left[(p_a + \varepsilon_a)\ln\left(1 + \frac{\varepsilon_a}{p_a}\right)\right] && \text{by linearity of exp.} \\
&= p_a \ln(p_a) + 0 + \mathbb{E}\left[(p_a + \varepsilon_a)\ln\left(1 + \frac{\varepsilon_a}{p_a}\right)\right] && \text{as } \mathbb{E}[\varepsilon_a] = 0 \text{ and } p_a \text{ is constant.}
\end{aligned}$$

Now, since N is so large, ε_a is tightly distributed around 0 and so $\left|\frac{\varepsilon_a}{p_a}\right| \ll 1$ almost always. Hence we can approximate the remaining expectation with Taylor's expansion of $\ln(1+x)$ for $|x| < 1$ as follows:

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a \ln(\hat{p}_a)\right] &\approx p_a \ln(p_a) + \mathbb{E}\left[(p_a + \varepsilon_a)\left(\frac{\varepsilon_a}{p_a} - \frac{\varepsilon_a^2}{2p_a^2} + \frac{\varepsilon_a^3}{3p_a^3} + O\left(\varepsilon_a^4\right)\right)\right] \\
&= p_a \ln(p_a) + \mathbb{E}\left[\varepsilon_a + \frac{\varepsilon^2}{2p_a} - \frac{\varepsilon^3}{6p^2} + O\left(\varepsilon_a^4\right)\right].
\end{aligned}$$

Since ε_a is normally distributed around 0 the expectation of its odd powers are all 0. Furthermore, since ε_a is small we shall ignore powers larger than 4. Hence,

$$\mathbb{E}\left[\hat{p}_a \ln(\hat{p}_a)\right] \approx p_a \ln(p_a) + \mathbb{E}\left[\frac{\varepsilon_a^2}{2p_a}\right].$$

So, since $\mathbb{E}[\varepsilon_a] = 0$, we have $\mathbb{E}[\varepsilon_a^2] = \mathrm{Var}[\varepsilon_a] = \sigma_a^2$ and hence we arrive at the following approximation

$$E\left[\hat{p}_a \ln(\hat{p}_a)\right] \approx p_a \ln(p_a) + \frac{\sigma_a^2}{2p_a}. \tag{3.3}$$

Now we consider the other term in our approximation for the variance, namely, $\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right]$. We again use the equivalence of $\hat{p}_a = p_a + \varepsilon_a$ so that we have,

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] &\approx \mathbb{E}\left[(p_a+\varepsilon_a)^2 \ln(p_a+\varepsilon_a)^2\right] \\
&= \mathbb{E}\left[(p_a+\varepsilon_a)^2 \left(\ln(p_a) + \ln\left(1+\frac{\varepsilon_a}{p_a}\right)\right)^2\right] \\
&= \mathbb{E}\left[(p_a^2 + 2p_a\varepsilon_a + \varepsilon_a^2)\left(\ln(p_a)^2 + 2\ln(p_a)\ln\left(1+\frac{\varepsilon_a}{p_a}\right) + \ln\left(1+\frac{\varepsilon_a}{p_a}\right)^2\right)\right].
\end{aligned}$$

Expanding further, and using linearity of expectation, we get,

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] \approx\ & \mathbb{E}\left[p_a^2 \ln(p_a)^2\right] + 2p_a \ln(p_a)^2 \mathbb{E}\left[\varepsilon_a\right] + \ln(p_a)^2 \mathbb{E}\left[\varepsilon_a^2\right] \\
&+ \mathbb{E}\left[(p_a^2 + 2p_a\varepsilon_a + \varepsilon_a^2)\left(2\ln(p_a)\ln\left(1+\frac{\varepsilon_a}{p_a}\right) + \ln\left(1+\frac{\varepsilon_a}{p_a}\right)^2\right)\right].
\end{aligned}$$

Once again, since $\varepsilon_A \sim N\left(0, \sigma_a^2\right)$, this reduces to,

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] \approx\ & p_a^2 \ln(p_a)^2 + 0 + \ln(p_a)^2 \sigma_a^2 \\
&+ \mathbb{E}\left[(p_a^2 + 2p_a\varepsilon_a + \varepsilon_a^2)\left(2\ln(p_a)\ln\left(1+\frac{\varepsilon_a}{p_a}\right) + \ln\left(1+\frac{\varepsilon_a}{p_a}\right)^2\right)\right].
\end{aligned}$$

We can again apply the Taylor expansion for $\ln(1+x)$ for $|x| < 1$ to get,

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] &\approx \ln(p_a)^2 (p_a^2 + \sigma_a^2) \\
&+ \mathbb{E}\left[(p_a^2 + 2p_a\varepsilon_a + \varepsilon_a^2)\left(2\ln(p_a)\left(\frac{\varepsilon_a}{p_a} - \frac{\varepsilon_a^2}{2p_a^2} + \frac{\varepsilon_a^3}{3p_a^3} + O\left(\varepsilon_a^4\right)\right) + \left(\frac{\varepsilon_a^2}{p_a^2} - \frac{\varepsilon_a^3}{p_a^3} + O\left(\varepsilon_a^4\right)\right)\right)\right].
\end{aligned}$$

Collecting terms we get,

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] &\approx \ln(p_a)^2 (p_a^2 + \sigma_a^2) \\
&+ \mathbb{E}\left[(p_a^2 + 2p_a\varepsilon_a + \varepsilon_a^2)\left(\frac{2\ln(p_a)}{p_a}\varepsilon_a + \frac{1-\ln(p_a)}{p_a^2}\varepsilon_a^2 + \frac{2\ln(p_a)-3}{3p_a^3}\varepsilon_a^3 + O\left(\varepsilon_a^4\right)\right)\right].
\end{aligned}$$

We can further expand the expression in the remaining expectation and then collect terms to get,

$$\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] \approx \ln(p_a)^2 (p_a^2+\sigma_a^2) + \mathbb{E}\left[2p_a\ln(p_a)\varepsilon_a + (1+3\ln(p_a))\varepsilon_a^2 + \frac{3+2\ln(p_a)}{3p_a}\varepsilon_a^3 + O\left(\varepsilon_a^4\right)\right].$$

Again, since $\varepsilon_A \sim N\left(0, \sigma_a^2\right)$ the odd powers of ε_a provide zero moments/expectation, and since ε_a is small we will ignore powers higher than 4 in our approximation. Therefore, our approximation reduces to,

$$\begin{aligned}
\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] &\approx \ln(p_a)^2 (p_a^2 + \sigma_a^2) + (1+3\ln(p_a))\mathbb{E}\left[\varepsilon_a^2\right] \\
&= \ln(p_a)^2 (p_a^2 + \sigma_a^2) + (1+3\ln(p_a))\sigma_a^2.
\end{aligned}$$

Thus we arrive at the following approximation,

$$\mathbb{E}\left[\hat{p}_a^2 \ln(\hat{p}_a)^2\right] \approx p_a^2 \ln(p_a)^2 + (1 + 3\ln(p_a) + \ln(p_a)^2)\sigma_a^2. \tag{3.4}$$

We can now use approximations 3.2 and 3.3 in approximation 3.1 to get an approximation for the variance as follows,

$$\begin{aligned}\operatorname{Var}[\hat{H}] &\approx \frac{1}{(\ln 2)^2} \sum_{a\in\mathcal{B}} \left(p_a^2 \ln(p_a)^2 + (1 + 3\ln(p_a) + \ln(p_a)^2)\sigma_a^2 - \left(p_a \ln(p_a) + \frac{\sigma_a^2}{2p_a} \right)^2 \right) \\ &= \frac{1}{(\ln 2)^2} \sum_{a\in\mathcal{B}} \left((1 + 2\ln(p_a) + \ln(p_a)^2)\sigma_a^2 - \frac{\sigma_a^4}{4p_a^2} \right).\end{aligned}$$

For large N, σ_a^4 is negligible so we can reduce this to the following,

$$\begin{aligned}\operatorname{Var}[\hat{H}] &\approx \frac{1}{(\ln 2)^2} \sum_{a\in\mathcal{B}} (1 + \ln(p_a))^2 \sigma_a^2 \\ &= \frac{1}{(\ln 2)^2} \sum_{a\in\mathcal{B}} (\ln e + \ln(p_a))^2 \sigma_a^2 \\ &= \frac{1}{(\ln 2)^2} \sum_{a\in\mathcal{B}} \ln(e p_a)^2 \sigma_a^2.\end{aligned}$$

Thus we arrive at the approximation,

$$\operatorname{Var}[\hat{H}] \approx \sum_{a\in\mathcal{B}} \left(\sigma_a \log_2(e p_a)\right)^2. \tag{3.5}$$

We also have an approximation for the expectation, by inserting 3.3 into 3.1, which gives,

$$\mathbb{E}[\hat{H}] \approx H - \sum_{a\in\mathcal{B}} \frac{\sigma_a^2}{2p_a \ln 2}. \tag{3.6}$$

Where H is the true entropy of the distribution defined by $\{p_a\}$. This shows that our plug-in entropy estimate will, on average, underestimate the entropy. If we sought a better estimate we would account for this. For our purposes this is not necessary as even with this under estimate, the variance will be too large for us to reliably distinguish the distribution from one with maximal entropy. To demonstrate this we will select a particular set of values $\{p_a\}$ that is close to the actual distribution for Z_5 and we will calculate the resulting values of $\mathbb{E}[\hat{H}]$ and $\operatorname{Var}[\hat{H}]$. I have chosen Z_5 somewhat arbitrarily, except to say that it is a byte position that would be of interest in single byte bias attacks. The results are similar for most other early bytes. The values used here for the $\{p_a\}$ are those observed in our experimental data collection. These, should be close to the correct values and are simply meant to be illustrative of the problem. Using our approximations the distribution provides the following values for the expectation and variance:

$$\begin{aligned}\mathbb{E}[\hat{H}] &\approx 7.999999406911917, \\ \operatorname{Var}[\hat{H}] &\approx 4.211233838743257 \times 10^{-12}.\end{aligned}$$

Even if we only consider a confidence interval that is approximately 1 standard deviation wide then this already encompasses the value 8:

$$\left(\mathbb{E}[\hat{H}] - \frac{1}{2}\sqrt{\mathrm{Var}[\hat{H}]}\ ,\ \mathbb{E}[\hat{H}] + \frac{1}{2}\sqrt{\mathrm{Var}[\hat{H}]}\right) \approx (7.9999984, 8.0000004) \qquad \text{(to 7 d.p.)}$$

In fact, reducing this confidence interval to exclude the value 8 would require $N > 2^{51}$. If we wished to use a confidence interval 4 standard deviations wide we would require $N > 2^{55}$. If we used an adjusted estimator to account for the plug-in entropy's bias to under estimate the true value, then we would require more samples (larger N). This would simply be to distinguish the distribution from the uniform distribution (i.e. to establish that there is some vulnerability), we would also wish to use the figures as a means to quantify the level of vulnerability so this would actually require even more samples. Given the computing resources available, this is not possible at this time. Therefore, we will not utilise the experimental entropy as a metric for vulnerability.

3.4 An Attacker's Perspective

An alternative approach is to consider the vulnerability analysis from the perspective of an attacker. This requires that we specify the strength of the attacker in question.

3.4.1 The Attacker

We shall consider ciphertext Only Attacks in the Broadcast setting. i.e. Eve only has access to ciphertexts from Alice, however, she can see multiple encryptions of the same plaintext using randomly selected secret keys. We also assume that Eve knows that Alice is using RC4 with randomly selected keys as the encryption system (as per Kerckhoffs' Principle).

If RC4 key-stream output were genuinely indistinguishable from a random stream then no matter how many encryptions she saw, Eve would never gain more information about the plaintext than she had to begin with. However, since RC4 has non-uniformities in its output, these can be used to inform Eve in her attempts to discover the plaintext.

The General Single Byte Attack

A general attack in this setting can be as follows. Recalling our notation, C_r is the value of the ciphertext output by Alice when sending an encryption of her r^{th} message byte m_r.

If Eve collects k ciphertexts from Alice, all with corresponding message byte m_r. Then she has made k observations of C_r. Eve knows that $C_r = Z_r \oplus m_r$ and so that $C_r \oplus m_r = Z_r$. Eve does not know the value of Z_r or m_r, however, we assume that she does know the distribution of Z_r (or a close approximation of it, established by similar methods to those we intend to use).

Eve knows that some $m \in \mathcal{B}$, is the message byte value m_r. So she only has 256 candidate values to consider, each of which permutes the distribution of Z_r differently under $\oplus$. Eve just needs to decide which value is the"best" guess which permutes the distribution so that it "matches" the observed distribution of the C_r. Eve has some freedom to decide what counts as a"better" guess or match and we will consider two specific methods in the following sections.

Eve may also wish to use the output from this process in different ways. She may be happy to have a sorted list of candidate m_r values as output, with the "best" match at the top. As long as

the correct m_r value is reliably close to the top of the list this might be sufficient for her needs. Perhaps she is looking for a plaintext that is written in a human tongue, it may be easy to sift a few candidates for the correct message and then stitch multiple bytes together. Alternatively, she may only want to use the "best" match from the possible values of m. Perhaps the plaintext has no known structure and so she needs to reliably receive the right answer to avoid having to do an exponential search through all possible multi-byte messages.

In measuring vulnerability we will consider both of these two settings in turn. For now we turn to describing some specific attacks by defining two ways Eve could seek to sort the candidate values for m_r by reference to the data.

The "Max-Peak" Single Byte Attack

We first consider a very naive heuristic approach to quantifying which value of $m \in \mathcal{B}$ is a better guess for the plaintext. Let $\{p_a = P[Z_r = a]\}$ be the correct distribution for Z_r and, let $\{\hat{p}^c_a = \hat{P}[C_r = a]\}$ be the observed distribution of the ciphertexts. Then define an order on the values of $m \in \mathcal{B}$ from best match to worst match, $(m^0, \ldots, m^{255})$, as follows:

First, using the oberved ciphertexts, define an order on the possible values of $a \in \mathcal{B}$,
$(a^0, \ldots, a^{255}),$ s.t. $\hat{p}^c_{a^0} \geq \hat{p}^c_{a^1} \geq \ldots \geq \hat{p}^c_{a^{255}}$.

Second let $a_{max} = \underset{a}{\operatorname{argmax}}(p_a),$ be the byte value that has the maximal probability in the RC4 stream output at position r.

Then, as our final step, we can define $(m^0, \ldots, m^{255}), by$
$m^i \coloneqq a^i \oplus a_{max}$.

In simple terms, this order looks at the experimental probabilities observed for the ciphertexts in order from largest to smallest. We then say that the best guess for m_r is the value of m that $\oplus$-maps the true peak for the RC4 key-stream output (found at a_{max}) to the position that was observed as the maximum in the cipher-stream, a^0. The second best guess is then the value that $\oplus$-maps the true peak to the second highest observed peak, a_1, and so on. This measure is essentially just trying to guess m_r by looking at where the maximum peak of the distribution might have been mapped to and giving more weight to higher peaks. This method therefore ignores a lot of other information that we might exploit about the observed distribution (e.g. does the suggested value for m^0 $\oplus$-map other features in the distribution in the most plausible way as well). However, because the distributions for the early RC4 output bytes often have isolated maximum peaks, this is still likely to be effective for these byte positions. If the distribution had many similar peaks, or a maximum that was close in value to many other parts of the distribution, then this method would be less successful. This attack is similar to that employed by Isobe et al. [8].

The Bayesian Single Byte Attack with Uniform Prior

A second, more thorough approach is to utilise the machinery of Bayesian analysis as per the single byte attack described in [3]. Again we use $\{p_a = P[Z_r = a]\}$ to denote the correct distribution for position r. We also let $H_a = \#$ of observations where $C_r = a$. We define h_a to be the experimentally observed value of H_a. We also now explicitly consider the message byte value to be

a random variable M_r whose actual value, m_r, Eve is uncertain of.
To ascertain the value of m which best fits the data available to Eve, we seek to maximise

$$P\left[M_r = m \mid \bigwedge_{a \in \mathcal{B}} H_a = h_a\right],$$

over the values of m. More generally we seek to sort the values of m by reference to these values. By Bayes' Theorem, this expression is equal to

$$\frac{P\left[\bigwedge_{a \in \mathcal{B}} H_a = h_a \mid M_r = m\right] \times P[M_r = m]}{\sum_{m' \in \mathcal{B}} P\left[\bigwedge_{a \in \mathcal{B}} H_a = h_a \mid M_r = m'\right] \times P[M_r = m']}$$

For the purposes of our analysis we will assume a prior distribution for $P[M_r = m]$ that is uniform i.e. we assume Eve has no information about the message byte values ahead of time. Since the value on the bottom of this fraction is independent of m and positive we can disregard it in seeking to sort the values of this expression. Also the value $P[M_r = m] = 1/256$, is constant for all m by assumption of a uniform prior, so we can also ignore this term. Thus we simply need to sort the values of m according to the values of

$$P\left[\bigwedge_{a \in \mathcal{B}} H_a = h_a \mid M_r = m\right].$$

Recalling that k is the total number of observations made by Eve, this can be written as,

$$P\left[\bigwedge_{a \in \mathcal{B}} H_a = h_a \mid M_r = m\right] = \frac{k!}{h_0! \cdot \ldots \cdot h_{255}!} \prod_{a \in \mathcal{B}} p_{a \oplus m}^{h_a}.$$

The multinomial coefficient at the start of this expression is again independent of m and positive so we can ignore this when sorting. Furthermore, since the log function is increasing on positive values we can reduce this all down to the problem of sorting the values of m according to the values of

$$\log\left(\prod_{a \in \mathcal{B}} p_{a \oplus m}^{h_a}\right) = \sum_{a \in \mathcal{B}} h_a \log\left(p_{a \oplus m}\right).$$

Given m and the values of $\{h_a\}$ and $\{\log p_a\}$, the values of this expression are relatively quick to compute. Sorting the values of m according to the resulting calculations is then a relatively simple step.
This describes a straight forward means to perform what we call the Bayesian single byte attack with a uniform prior.

3.4.2 Measuring RC4's Vulnerability to Our Attacker

We have defined two specific single byte attacks at Eve's disposal. We are now ready to describe how we will measure the vulnerability of the RC4 cipher is to these attacks. It is worth noticing

that whilst the attacker is relatively weak in our setting, only having access to ciphertexts; given an unlimited number of encryptions, these attacks may be able to expose any message byte in an RC4 byte position that is non-uniform. If the attacker knows the true distributions for Z_r then the only external limit on the attacker is how many encryptions she can collect. Furthermore, the more vulnerable a byte position is to an attack, the fewer encryptions we'd expect to be required for the attack to be "successful." This informs our measures of vulnerability. As mentioned we consider two settings, first, a setting where an attacker only seeks that the attack will give them a sorted list of candidates for m_r such that m_r is reliably near the top of the list. Second, a setting where the attacker only cares about the top candidate for m_r being correct.

n-shot Guessing

In this setting the attacker is concerned with collecting enough ciphertexts so that they can be confident that their attack will produce an ordered list that contains the correct message byte within some pre-defined number of places from the top of the list, n. We can quantify the vulnerability of the particular byte position of RC4 output in this setting by the number of ciphertexts required to reduce the average placement of the correct message value to be less than n. For a less vulnerable byte position an attacker will require more ciphertexts to get enough information to successfully place the correct message byte value within the top n candidates on average. In practice an attacker would likely seek a low value of n. We will perform our analysis with $n = 3$ and $n = 30$ for comparison.

One means of collecting data for this analysis would be to perform attacks with different numbers of ciphertexts repeatedly, using random keys, and measuring the average position of the correct message byte and observing when this falls below n. Unfortunately, this is computationally infeasible at this time as it would require us to repeatedly collect similar amounts of data as we did for our bias observation phase and we have not got sufficient computing time available for this purpose. As an alternative we will simulate this process. We do this by using our bias collection data to estimate the true distribution of the RC4 outputs. We will avoid the requirement to collect actual ciphertext samples and directly sample our histogram data. We simulate the repeated collection of a fixed number of ciphertexts by perturbing the distribution with normally distributed noise with approximately the correct standard deviation as would be seen in sampling the fixed number of ciphertexts. We ensure this noise sums to zero across the whole distribution. In doing this we are using the multivariate normal distribution as an approximation to the multinomial distribution. This is justified because the sample sizes will be large relative to the probabilities. We then use the resulting simulated ciphertext data to perform the attack and collect the results as though this were a genuine sample. We then perform this repeated sampling simulation for different numbers of ciphertexts to see how the average position of the correct result changes and then we can determine at what point this value falls below n for the values $n = 3$ and 30. This will only give us an estimate for the real figures, but this should be sufficient for the purposes of considering the vulnerability to our attacks.

In Algorithm 4 we have included pseudo code for the procedure to collect the necessary data for the n-shot setting using the "max-peak" attack. We implemented this in Matlab (2015-a).

Input: $\boldsymbol{p}$, a vector containing the correct key-stream distribution of Z_r; S, the number of simulated attacks to be run in collecting our averages.
Output: **averages**, a vector of the average number of guesses required for each ciphertext count simulated.

begin

Select random value for m_r; // This value will make no difference to the results

$\boldsymbol{d} \leftarrow \boldsymbol{p}$ with indices permuted by $\oplus$-ing with m_r; // Generate the correct ciphertext distribution of $Z_r \oplus m_r$

for *Number of sampled ciphertexts* $= t = 2^9, 2^{10}, \ldots, 2^k$ **do**

$\sigma \leftarrow \sqrt{\frac{\frac{1}{256}\cdot(1-\frac{1}{256})}{t}}$; // approx. value of the standard deviation for sampling noise.

$\boldsymbol{Q} \leftarrow \text{nullspace}(\{1\}^{256})$; // $\boldsymbol{Q}$ is a 256×255 matrix whose columns are an orthonormal basis for the null space of the 1 by 256 ones vector.

GaussianNoise $\leftarrow \text{randn}(255, S)$; // $255 \times S$ matrix of independent $N(0,1)$ distributed noise.

SimMultinomialNoise $\leftarrow \sqrt{\frac{256}{255}} \times \sigma \times \boldsymbol{Q} \times$ **GaussianNoise**; // $256 \times S$ matrix of scaled Gaussian noise to simulate noise in multinomial sampling of ciphertexts with t samples. Each column sums to 0 thanks to the inclusion of the null space matrix Q.

for *simulatedAttackNo* $= 1, \ldots, S$ **do**

synthDistribution $\leftarrow \boldsymbol{d} +$ **SimMultinomialNoise**$(:, \textit{simulatedAttackNo})$; // Add the noise to the true ciphertext distribution to get the simulated observed ciphertext distribution.

noOfGuesses $\leftarrow$ getNoGuessesMaxPeakAttack(**synthDistribution**, m_r); // Run the attack and check how many guesses are required to get the correct message byte value, m_r.

totalNoOfGuesses $\leftarrow$ *totalNoOfGuesses* + *noOfGuesses*;

end

averages$[t] \leftarrow \frac{\textit{totalNoOfGuesses}}{S}$; // Calculate the average no. of guesses for this number of observed ciphertexts.

end

return averages;

end

Algorithm 4: Outputs estimated figures for the average number of guesses required by the "max-peak" attack to get the correct message byte when provided different numbers of sampled ciphertexts. We used our experimental data to generate the input $\boldsymbol{p}$. We set $S = 30{,}000$ as with this value we observed very little variation in the results on repeated runs of the algorithm. The range of values for t was selected to get the full range of averages, from roughly 128 down to almost 1.

1-shot Guessing

In this setting the attacker only cares about whether their attack produces the correct message byte as the first guess, any subsequent guesses are treated as useless. In reality this is the more likely situation as, in general, an attacker may have no way to confirm that their guess is correct, in which case having multiple guesses is not necessarily useful anyway. In this setting it is natural that the attacker would be concerned with how confident they can be that the guess resulting from their attack is correct. Therefore, as a means to quantify vulnerability from this attacker's perspective we can ask, how many ciphertexts does the attacker need before they can have $q\%$ that their guess is correct? Depending on the context, the attacker might require different values for q. We will consider $q = 50$ and 99.
Again, due to the data requirements, collecting real attack data for this analysis will not be possible in our case, however, as before, we can simulate the results of collecting different quantities of ciphertexts and use these samples to estimate the probability that the attack gives the correct answer.
In algorithm 5 we have described the process used for collecting this data, again we only show the "max-peak" case as an example. The outer loop for this algorithm is essentially the same as that of algorithm 4 and so when implementing the algorithms it made sense to combine them both.

This concludes the description of our methodology for identifying and quantifying biases in the RC4 stream, collecting the required data, and then analysing the vulnerability of the cipher introduced by these biases. We now move on to report on to the observations that we made by following these steps.

Input: $\boldsymbol{p}$, a vector containing the correct key-stream distribution of Z_r; S, the number of simulated attacks to be run in collecting our averages.
Output: **precentages**, a vector of the average success rate on the first guess for each ciphertext count simulated.

begin

Select random value for m_r; // This value will make no difference to the results

$\boldsymbol{d} \leftarrow \boldsymbol{p}$ with indices permuted by $\oplus$-ing with m_r; // Generate the correct ciphertext distribution of $Z_r \oplus m_r$

for *Number of sampled ciphertexts* $= t = 2^9, 2^{10}, \ldots, 2^k$ **do**

$\sigma \leftarrow \sqrt{\frac{\frac{1}{256}\cdot(1-\frac{1}{256})}{t}}$; // approx. value of the standard deviation for sampling noise.

$\boldsymbol{Q} \leftarrow \text{nullspace}(\{1\}^{256})$; // $\boldsymbol{Q}$ is a 256×255 matrix whose columns are an orthonormal basis for the null space of the 1 by 256 ones vector.

GaussianNoise $\leftarrow \text{randn}(255, S)$; // $255 \times S$ matrix of independent $N(0,1)$ distributed noise.

SimMultinomialNoise $\leftarrow \sqrt{\frac{256}{255}} \times \sigma \times \boldsymbol{Q} \times$ **GaussianNoise**; // $256 \times S$ matrix of scaled Gaussian noise to simulate noise in multinomial sampling of ciphertexts with t samples. Each column sums to 0.

for *simulatedAttackNo* $= 1, \ldots, S$ **do**

synthDistribution $\leftarrow \boldsymbol{d} +$ **SimMultinomialNoise**$(:, \textit{simulatedAttackNo})$; // Add the noise to the true ciphertext distribution to get the simulated observed ciphertext distribution.

success $\leftarrow$ getSuccessOfMaxPeakAttack(**synthDistribution**, m_r); // Run the attack and output 1 if we get the correct message byte value, m_r, on the first guess, output 0 otherwise.

totalNoOfSuccesses $\leftarrow$ *totalNoOfSuccesses* $+$ *success*;

end

precentages$[t] \leftarrow \frac{\textit{totalNoOfSuccesses}}{S}$; // Calculate the percentage of successes on the first guess for this number of observed ciphertexts.

end

return precentages;

end

Algorithm 5: Outputs estimated figures for the percentage of successes by the 1-shot "max-peak" attack in guessing the correct message byte when provided different numbers of sampled ciphertexts. We used our experimental data to generate the input $\boldsymbol{p}$. We set $S = 30{,}000$ as with this value we observed very little variation in the results on repeated runs of the algorithm. The range of values for t was selected to get the full range of percentages, from roughly 1/256 up to almost 1.

Chapter 4

Results

The following chapter presents the results obtained through our work.

4.1 Single Byte Biases

We collected histograms of output bytes for the first 512 output bytes for $N = 2^{43.21}$ randomly selected RC4 keys.

4.1.1 Biases in the First 256 Bytes

We consider each previously identified bias in turn and decide whether a) our data confirm the existence of a bias or not, and b) whether our data confirm the theoretically derived value for the bias. We provide the experimental value with a 99.99% confidence range.

Single Byte Bias 1

The probability (over a random choice of key) that the second byte of key-stream output by RC4 is equal to 0 is approximately 1/128. i.e $b_0^2 \approx 256$

The Null Hypothesis is that $Pr[Z_2 = 0] = 1/256 = p_{H_0}$. The Z-score is

$$\frac{\hat{p} - p_{H_0}}{\sigma_{H_0}} = \frac{\hat{p} - p_{H_0}}{\sqrt{p_{H_0} \cdot (1 - p_{H_0})/N)}} = 199,968.$$

Using a two sided confidence level of 99.99% (Z score threshold of 3.891) we therefore can reject the null hypothesis that $Pr[Z_2 = 0] = 1/256$.
Now we provide a 99.99% confidence interval for the value of b_0^2:

$$256.323 \leq b_0^2 \leq 256.337.$$

This is equivalent to,

$$\frac{1}{128} + \frac{0.323}{256^2} \leq P[Z_2 = 0] \leq \frac{1}{128} + \frac{0.337}{256^2}.$$

The theoretically proposed figure for $P[Z_2 = 0]$ of 1/128 therefore approximately correct and the data suggests that this is a slight underestimate.

Single Byte Bias 2

$b_0^3 = 0.351089$ and $b_0^4, b_0^5, \ldots, b_0^{255}$ is a decreasing sequence with terms that are bounded as follows

$$0.242811 \leq b_0^r \leq 1.337057.$$

The experimentally observed values for these probabilities with $N \approx 2^{43.21}$ samples for each byte position are shown in Figure 4.1 (we have not included the confidence interval as these are visually indistinguishable from the plotted data).[1]

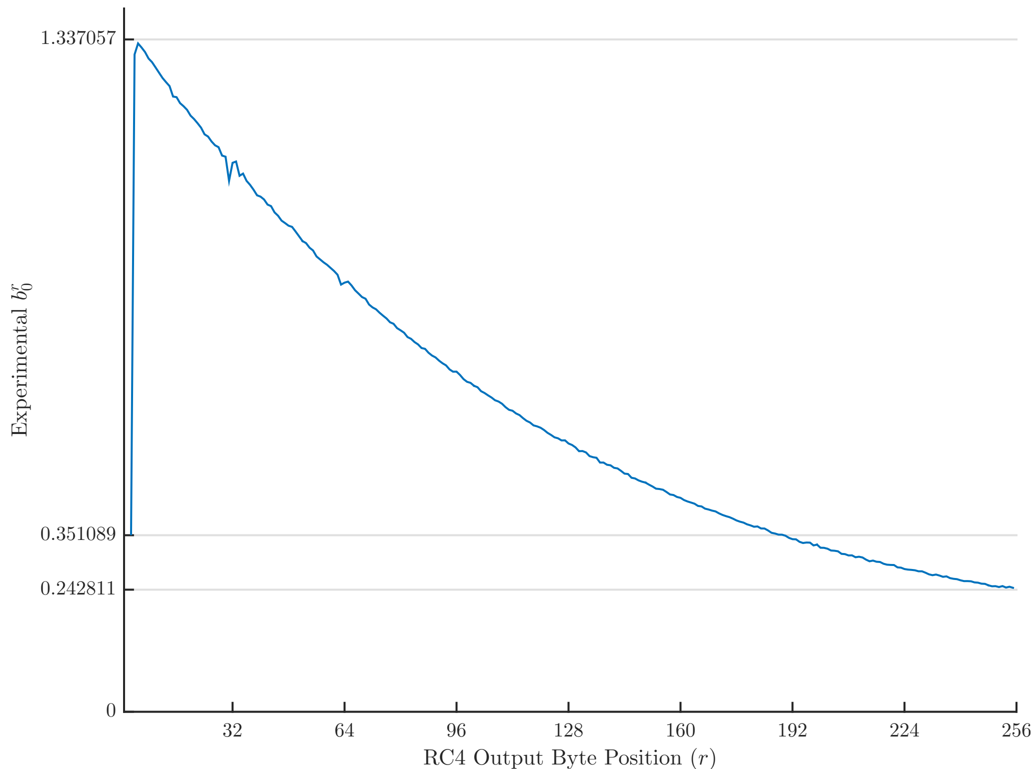


Figure 4.1: The experimentally observed values for b_0^r defined by $Pr[Z_r = 0] = \frac{1}{256} + \frac{b_0^r}{256^2}$ for $3 \leq r \leq 255$.

The Z-scores derived from the data for $3 \leq r \leq 255$ all provide sufficient evidence to reject the null hypothesis of uniformity with a 99.99% confidence level. The graph shows that the observed data is consistent with the pattern of biases proposed with one caveat. Namely, that the data for $4 \leq r \leq 255$ is not monotone decreasing. There are several areas of deviation visible, where the values b_0^r increase with r. These sets of values are $r = 4, 5$, $r = 31, 32, 33$, $r = 34, 35$, and $r = 64, 65$. For the specific values for $r = 3, 4, 255$ we can compare the proposed values of b_0^r to the observed

[1] We have chosen to plot our discrete bias data with lines joining our data points. These should clearly not be read to interpolate between the discrete x-axis values. We use the lines as a means to make it easier to read from the graph where the more dramatic changes in the bias values occur. Often there are isolated peaks in our data and these would be harder to identify without the neighbouring lines.

confidence intervals. The intervals are,

$$0.348 \le b_0^3 \le 0.358,$$
$$1.302 \le b_0^4 \le 1.312, \text{ and}$$
$$0.242 \le b_0^{255} \le 0.252.$$

The first and last intervals include the postulated values for the bias. However, the second interval does not.

Single Byte Bias 3

For keys of length l bytes, the lth byte of key-stream output is biased towards $256 - l$ with a bias greater than $1/256^2$. i.e. $b^l_{256-l} \ge 1$.

For the implementation of RC4 in TLS the key length l is 16 bytes. The null hypothesis in this case is that $P[Z_{16} = 240] = 1/256$, i.e. $p_{H_0} = 1/256$. The 2 sided Z-score threshold for a 99.99% confidence level is ± 3.891. The observed Z-score was 711 so we have sufficient evidence to reject the null hypothesis of uniformity. In fact a 99.99% confidence interval for the value of $b^l_{256-l} = b^{16}_{240}$ is

$$9.114 \le b^{16}_{240} \le 9.124.$$

So the bias is in fact roughly 9 times greater than that proposed in the literature.

Single Byte Bias 4

For keys of length 16 bytes (as used in TLS) the first key-stream byte is biased away from 129. i.e. $b^1_{129} < 0$

To test the existence of the bias in this case the null hypothesis is that $Pr[Z_1 = 129] = 1/256$, $p_{H_0} = 1/256$. From the data we get a Z-score of $-1,343$, so we have sufficient evidence to reject the hypothesis in favour of the bias at a 99.99% confidence level. A 99.99% confidence interval for the correct value of b^1_{129} is

$$-1.726 \le b^1_{129} \le -1.716$$

Single Byte Bias 5a

For positions $1 \le r \le 256$ in the key-stream output, there is a bias towards value r. i.e. $b^r_{r \bmod 256} > 0$

The data provides sufficient evidence to reject the Null Hypothesis of uniformity for $1 \le r \le 256$. However, the biases are not all towards r. In the cases of $r = 1, 2, 256$ the biases are negative. The observed values for the biases for all r are plotted in Figure 4.2.

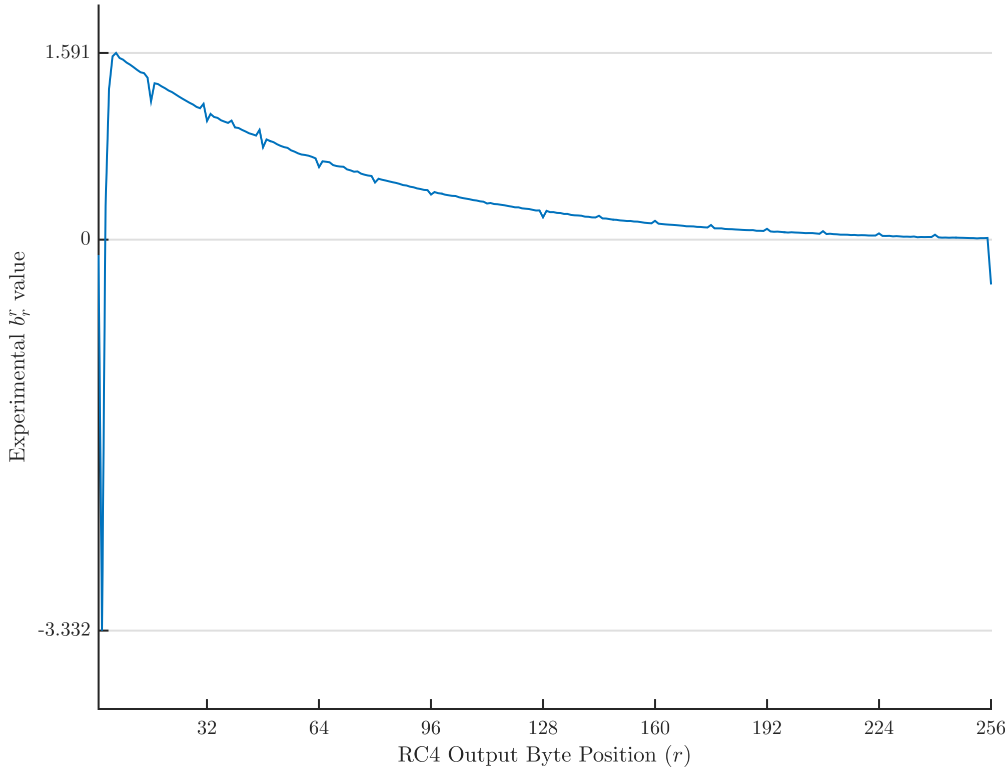


Figure 4.2: The experimentally observed values for b_r^r defined by $Pr[Z_r = r] = \frac{1}{256} + \frac{b_r^r}{256^2}$ for $1 \leq r \leq 256$.

Single Byte Bias 5b

For r that is a multiple of (the key length) 16, there is also a bias towards $256 - r$. i.e $b_{256-r}^r > 0$

The data is sufficient to reject the Null Hypothesis of uniformity for all r except $r = 89, 90, 91, 92$. The data are shown in Figure 4.3 and we can see that there is an apparent positive bias for most the values of $r \leq 88$ (even those which are not multiples of 16) as well as those $r \leq 128$ which are multiples of 16. Furthermore, the pattern of biases before and after $r = 128$ are very different (for the multiples and the non multiples of 16). In fact the probabilities for $128 < r < 256$ appear almost constant and slightly below $1/256$, it appears then that there is in fact no positive bias for $r > 128$ when r is a multiple of 16.

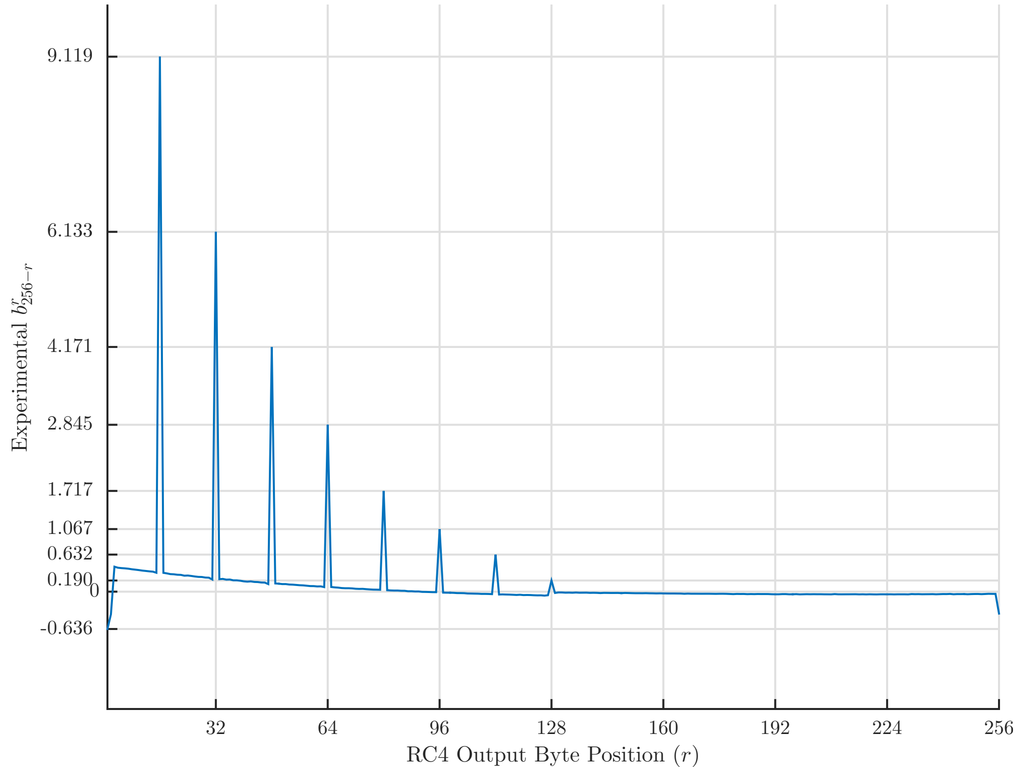


Figure 4.3: The experimentally observed values for b^r_{256-r} defined by $Pr[Z_r = 256-r] = \frac{1}{256} + \frac{b^r_{256-r}}{256^2}$ for $1 \leq r \leq 256$.

4.1.2 Biases in the Second 256 Bytes

We also sought to extend the existing literature by exploring the second page of RC4 output for single byte biases.

Non-Uniformities in the Distribution of Z_{257}

The experimental distribution of the first byte of the second page ($r = 257$) reveals some non-uniformities. Figure 4.4 shows the observed bias values. The most striking bias is for $a = 0$. The data is sufficient to reject a null hypothesis of uniformity for $P[Z_{257} = 0]$ at a 99.99% confidence level and a 99.99% confidence interval for b_0^{257} is

$$0.342 \leq b_0^{257} \leq 0.352.$$

From the graph we also see a general slope in the biases, culminating in a hump, which results in a preference towards higher byte values over lower ones. We shall see that a similar (though shallower) slope is a general feature of the distributions of bytes on the second page until roughly the 128th position on the page ($r = 384$).

Non-Uniformities in the Distributions of Z_{258} and Z_{259}

Figure 4.5 shows the observed bias values for $r = 258$. We can see from the scale that the biases

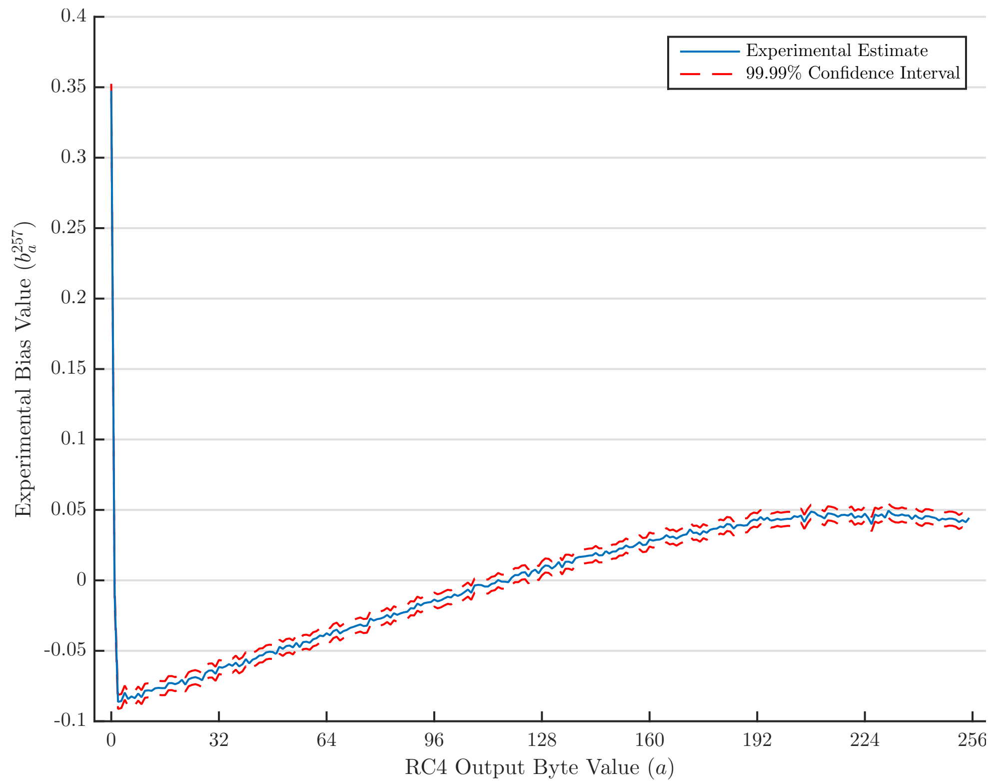


Figure 4.4: The experimentally observed values for b_a^{257} for $a \in \mathcal{B}$ with a 99.99% confidence interval.

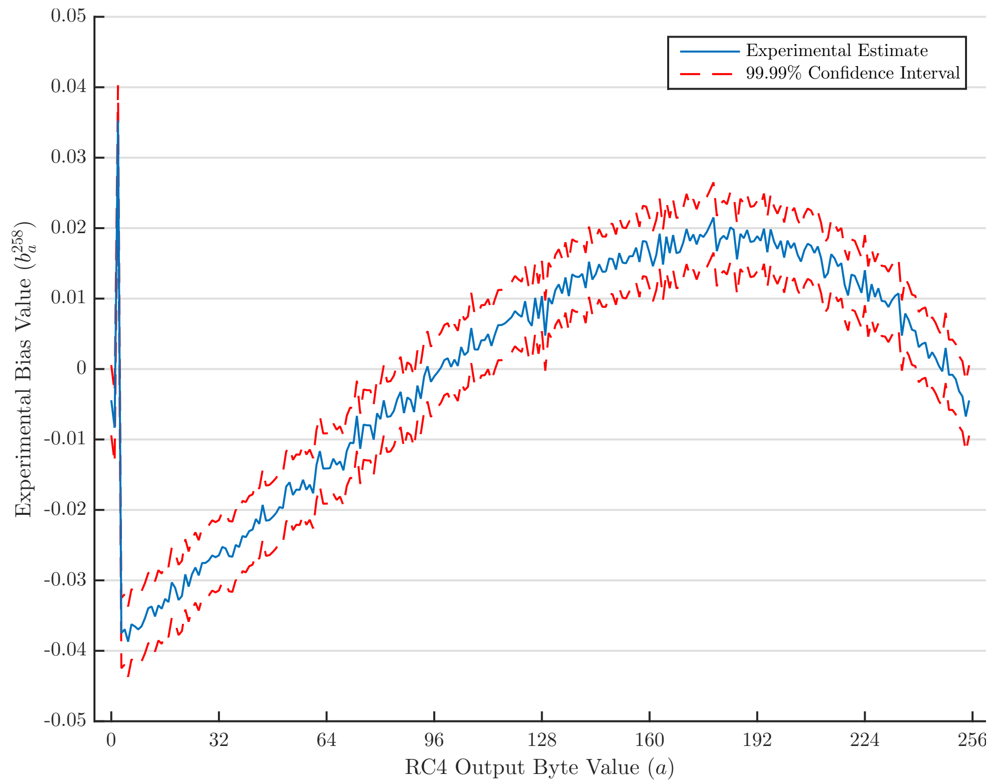


Figure 4.5: The experimentally observed values for b_a^{258} for $a \in \mathcal{B}$ with a 99.99% confidence interval.

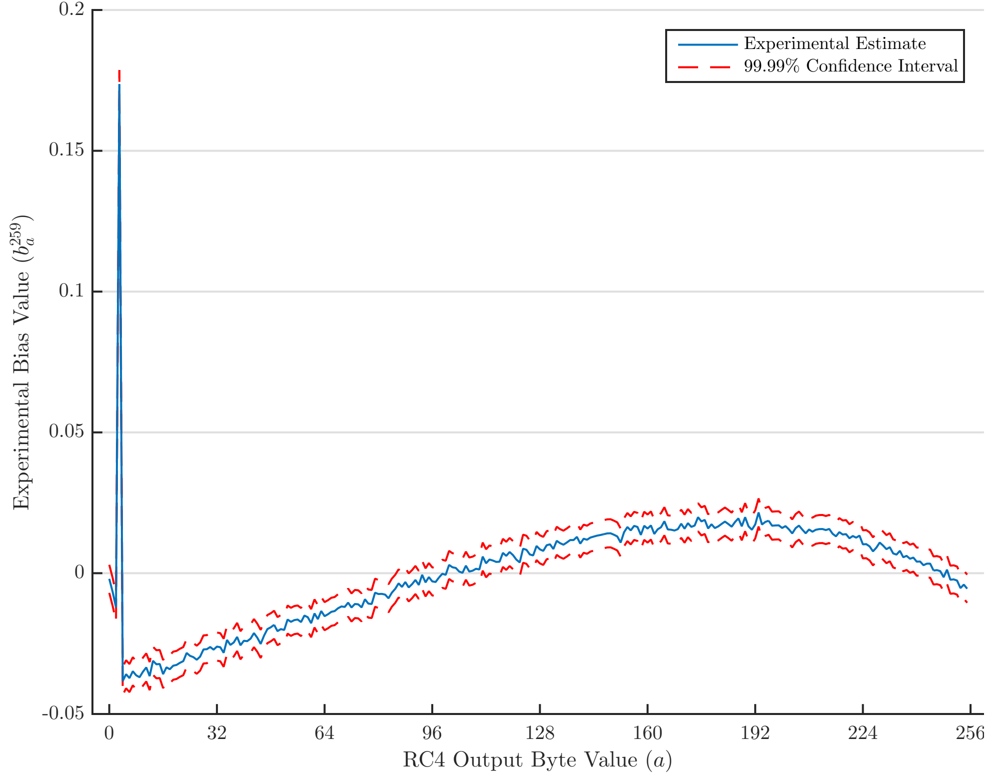


Figure 4.6: The experimentally observed values for b_a^{259} for $a \in \mathcal{B}$ with a 99.99% confidence interval.

are smaller than at $r = 257$. However, we again see the slope up to a hump for higher values of a and a peak on the far left. The peak on the left occurs at $a = 2$. The 99.99% confidence interval for b_2^{258} is

$$0.030 \leq b_2^{258} \leq 0.040.$$

Figure 4.6 shows the observed bias values for $r = 259$. Again we see a peak followed by a slope and hump. In this case the peak is at $a = 3$ and is larger than at $r = 258$, here the confidence interval is

$$0.169 \leq b_3^{259} \leq 0.179$$

At this point we might conjecture that future bytes will have a similar peak, slope & hump pattern. In fact we find something slightly different from here onwards on the second page.

Biases $b^r_{r \bmod 256}$ and $b^r_{r+1 \bmod 256}$ on the Second Page

In Figure 4.7 we show the biases b_a^r for various values of $260 \leq r \leq 336$. Whilst we can see that there is still a sloping feature up to a hump, there is no longer a strong peak to the left. Instead, as r increases we see that the graph becomes increasingly uniform. In all of the graphs there appears to be a discontinuity at $a = r \bmod 256$. To the left of this position the graph is much closer to uniform, then at $a = r + 1 \bmod 256$ There is a sudden drop. Graphically it appears that there is a wave rolling through the distributions at $a = r \bmod 256$ which washes away a large amount of the non-uniformity. We can dig a little deeper to observe the position dependent biases $b^r_{r \bmod 256}$

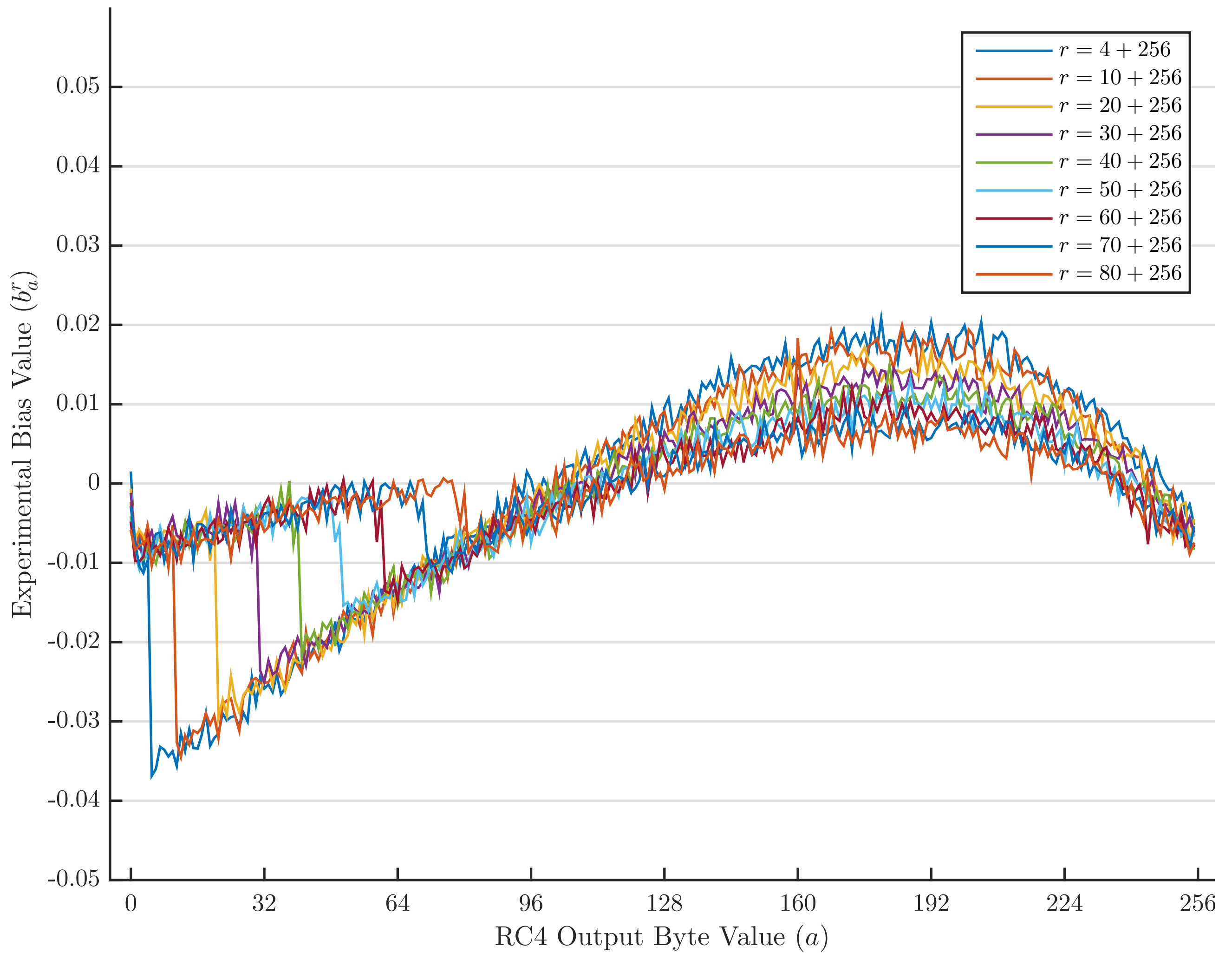


Figure 4.7: The experimentally observed values for b_a^r for $a \in \mathcal{B}$, and various values of r.

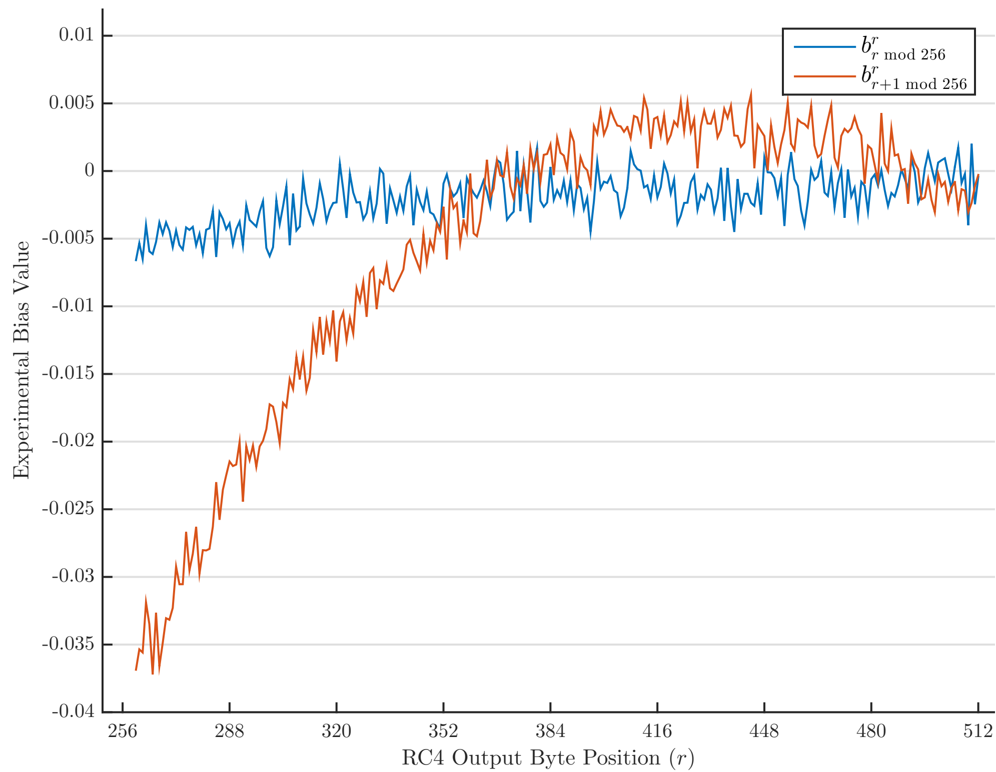


Figure 4.8: The experimentally observed values for $b^r_{r \bmod 256}$ and $b^r_{r+1 \bmod 256}$ for $260 \leq r \leq 512$.

and $b^r_{r+1 \bmod 256}$ to illustrate this. Figure 4.8 shows these values. The difference between the two lines is the height of the "wave-front" for each value of r. We can see that the differences flip sign at around $r = 352$ ($r \bmod 256 = 96$), which is also where we can see that the "wave-front" in figure 4.7 will "run into" the sloping part of the distribution. After this point we can anticipate a much more uniform distribution and this is what we find. Figure 4.9 shows the distributions for $r = 357, 407, 457$, and 507. We can see that their range of biases are much smaller and the distributions are harder to distinguish from the uniform distribution. However, we do still see the hints of a slope followed by a hump until at least $r = 407$. Beyond this there is one other sequence of strong biases that is apparent from the data.

Biases $b^r_{2r \bmod 256}$ for $r \equiv 0 \bmod 16$ on the Second Page

In figure 4.10 we show the distribution of biases at several positions r where $r = 0 \bmod 16$. As well as the previously described position related jump at $a = r \bmod 256$ we also see a strong bias at $a = 2r \bmod 256$. Figure 4.11 shows the biases $b^r_{2r \bmod 256}$ for $257 \leq r \leq 512$. The strong peaks clearly occur only where $r \equiv 0 \bmod 16$ and they appear to degrade and disappear by $r = 384 = 128 + 256$. Since such a peak was not observed for other values of r we hypothesise that this is a key length dependent bias.

4.2 Double Byte Biases

We collected histograms of output bytes for selected triplets (i, a, b) with $N = 2^{45}$.

4.2.1 Fluhrer-McGrew Biases

We began by checking the data to see whether it provides sufficient evidence for the existence of the Fluhrer-McGrew double byte biases in table 4.1, and if so to check whether the data provided a confidence interval which included the proposed values for the biases. We had data which allowed us to check all of the biases. The evidence was sufficient to reject uniformity in all of these cases at a 99.99% confidence level and the 99.99% confidence intervals all included the values predicted by Fluhrer and McGrew. In table 4.1 we have include a selection of the confidence intervals as examples. In figures 4.12 and 4.13 we show the observed bias values and confidence intervals for the Fluhrer-McGrew events that we observed.

4.3 Vulnerability Analysis

We now turn to our analysis of the vulnerabilities introduced to the RC4 cipher by the single byte non-uniformities that we have observed. We will use the metrics laid out earlier, this involves two settings (n-shot and 1-shot) each with two attacks ("max-peak" and Bayes). Figures 4.14 and 4.15 show the results from our synthetic sampling analyses for each setting respectively. Several observations stand out from the data.

First, within each setting, like-for-like comparison clearly shows that the Bayesian attack is "stronger" than the "Max-Peak" attack. Here we say stronger in the sense that it requires fewer ciphertexts to achieve equivalent success (whether success is measured by average number of required guesses or the percentage of correct first guesses). This is as expected, but the exact scale

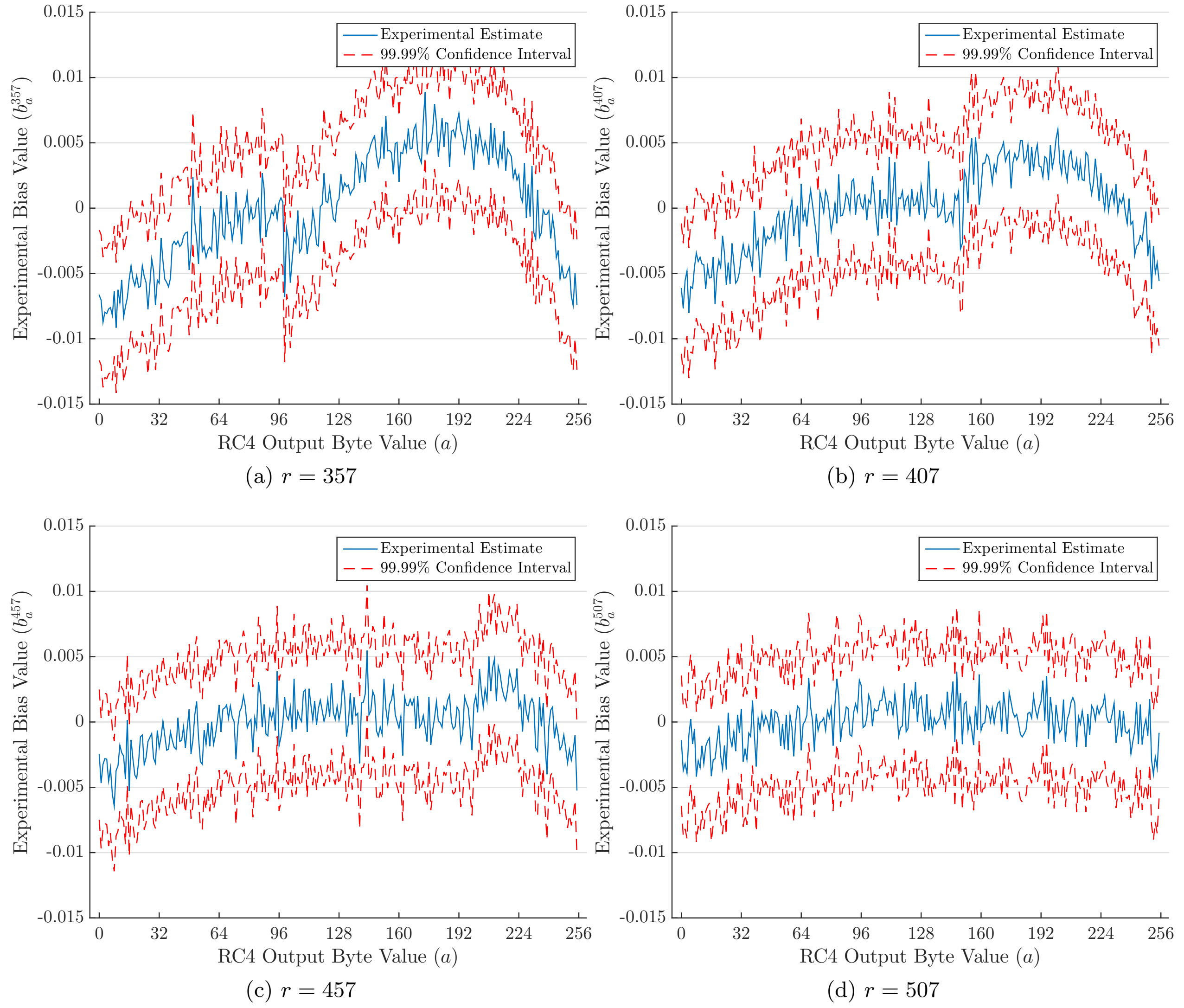


Figure 4.9: The experimentally observed values for b_a^r for $a \in \mathcal{B}$ and $r = 357, 407, 457$, and 507 with 99.99% confidence intervals.

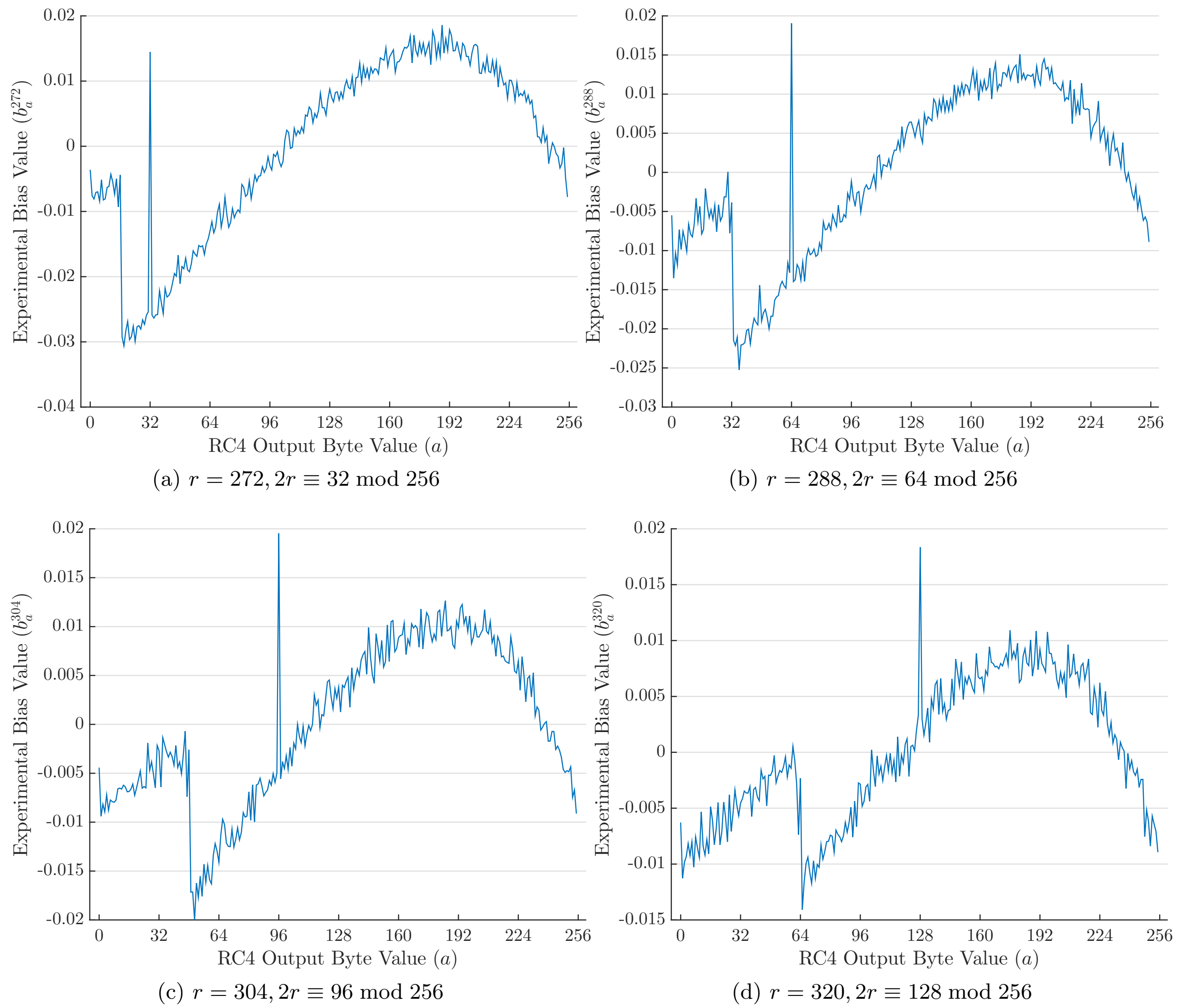


Figure 4.10: The experimentally observed values for b_a^r for $a \in \mathcal{B}$ and $r = 272, 288, 304,$ and 320 which are all multiples of 16.

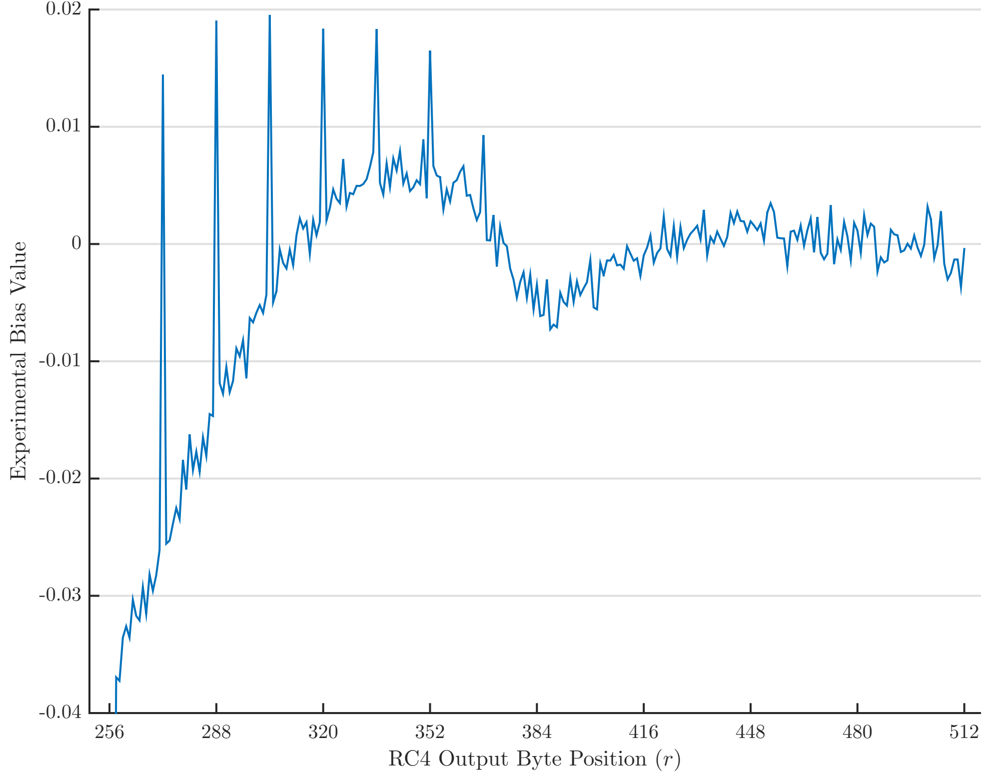
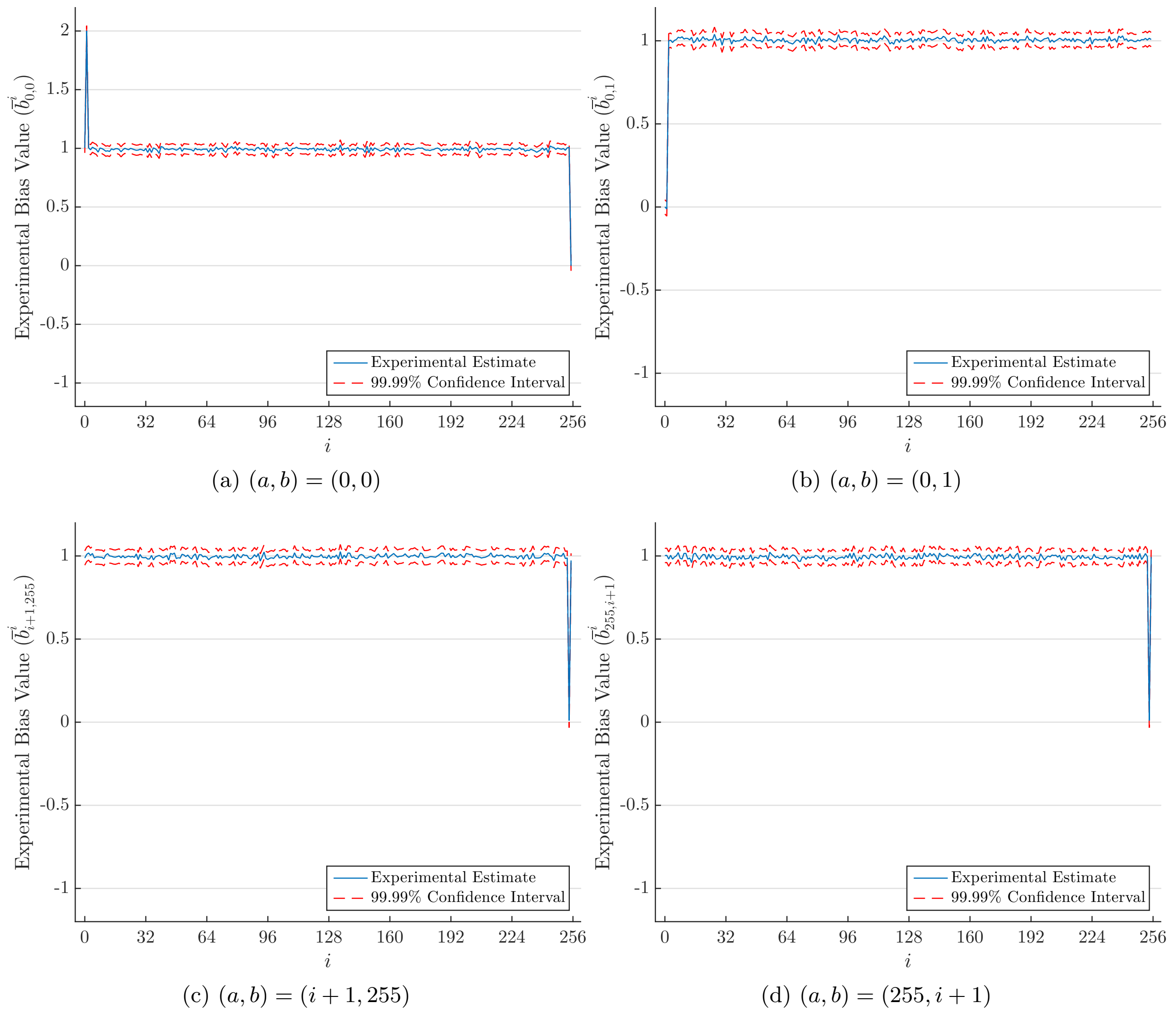


Figure 4.11: The experimentally observed values for $b^r_{2r \bmod 256}$ for $257 \leq r \leq 512$. We have cut off the trough at $r = 257$ as this is a special case which is not relevant.

Value(s) of $(a,b) = (Z_r, Z_{r+1})$	Value(s) of $i = r \bmod 256$	Approximate value of $\bar{b}^i_{a,b}$	Selected Confidence Interval
$(0,0)$	$i = 1$	$+2$	$1.959 \leq \bar{b}^1_{0,0} \leq 2.045$
$(0,0)$	$i \neq 1, 255$	$+1$	$0.961 \leq \bar{b}^0_{0,0} \leq 1.047$
$(0,1)$	$i \neq 0, 1$	$+1$	$0.957 \leq \bar{b}^2_{0,1} \leq 1.043$
$(i+1, 255)$	$i \neq 254$	$+1$	$0.945 \leq \bar{b}^0_{1,255} \leq 1.031$
$(255, i+1)$	$i \neq 254$	$+1$	$0.960 \leq \bar{b}^0_{255,1} \leq 1.046$
$(255, i+2)$	$i \neq 253$	$+1$	$0.948 \leq \bar{b}^{254}_{255,0} \leq 1.035$
$(129, 129)$	$i = 2$	$+1$	$0.949 \leq \bar{b}^2_{129,129} \leq 1.034$
$(255, 255)$	$i \neq 254$	-1	$-1.062 \leq \bar{b}^2_{255,255} \leq -0.976$
$(0, i+1)$	$i \neq 0, 255$	-1	$-1.070 \leq \bar{b}^1_{0,2} \leq -0.984$

Table 4.1: The Fluhrer-McGrew double byte biases first predicted in [6] with a selection of our observed confidence intervals.



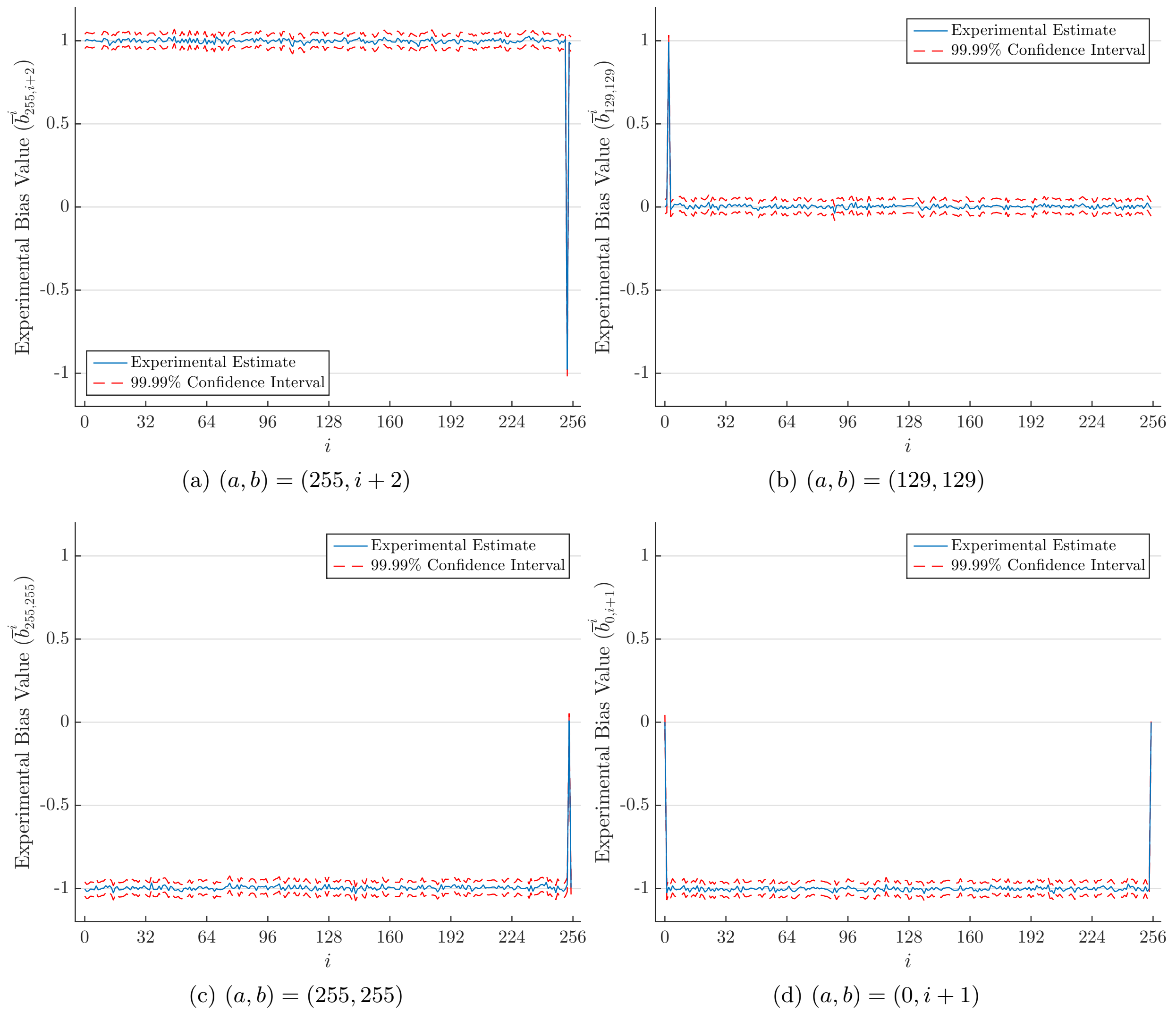
(a) $(a, b) = (0, 0)$

(b) $(a, b) = (0, 1)$

(c) $(a, b) = (i + 1, 255)$

(d) $(a, b) = (255, i + 1)$

Figure 4.12: The experimentally observed values of $\overline{b}^i_{a,b}$ for selected values of (a, b) and for $i \in \mathcal{B}$. They also include a 99.99% confidence interval.



(a) $(a, b) = (255, i + 2)$

(b) $(a, b) = (129, 129)$

(c) $(a, b) = (255, 255)$

(d) $(a, b) = (0, i + 1)$

Figure 4.13: The experimentally observed values of $\overline{b}^i_{a,b}$ for selected values of (a, b) and for $i \in \mathcal{B}$. They also include a 99.99% confidence interval.

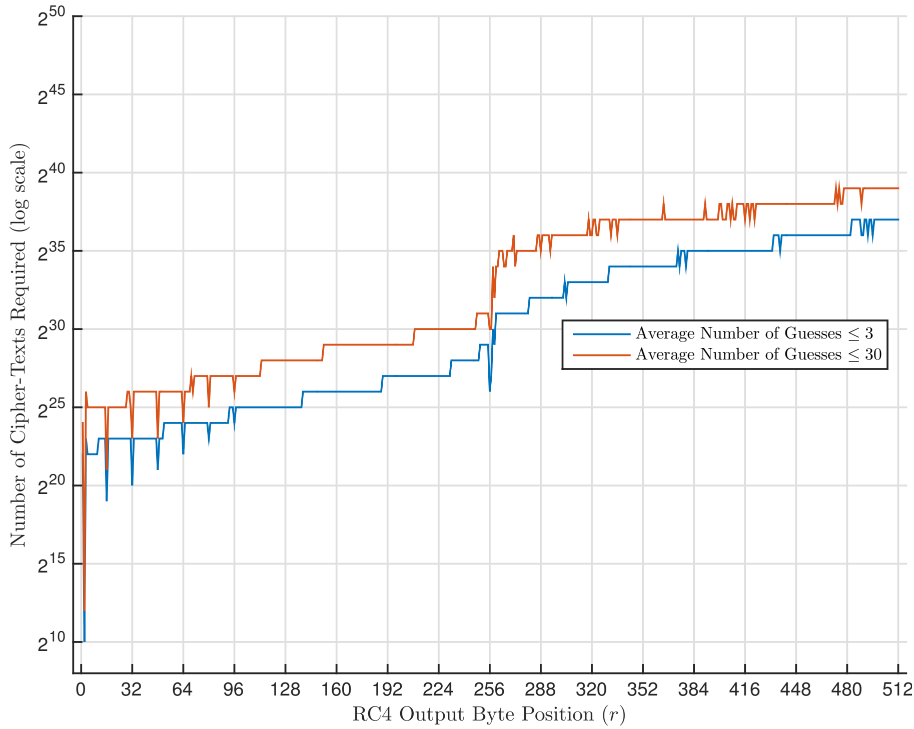
of the difference might lead an attacker to use one attack over the other depending on the total running time resulting from the number of ciphertexts sought and the algorithm selected.

Second, as we would expect from the size of the observed biases, the second page of output is generally less vulnerable than the first page. Roughly speaking achieving equivalent success on the second page appears to require increasing the number of ciphertexts by a factor of approximately 2^{10} with the Bayesian attack and 2^{12} with the "Max-Peak" attack.

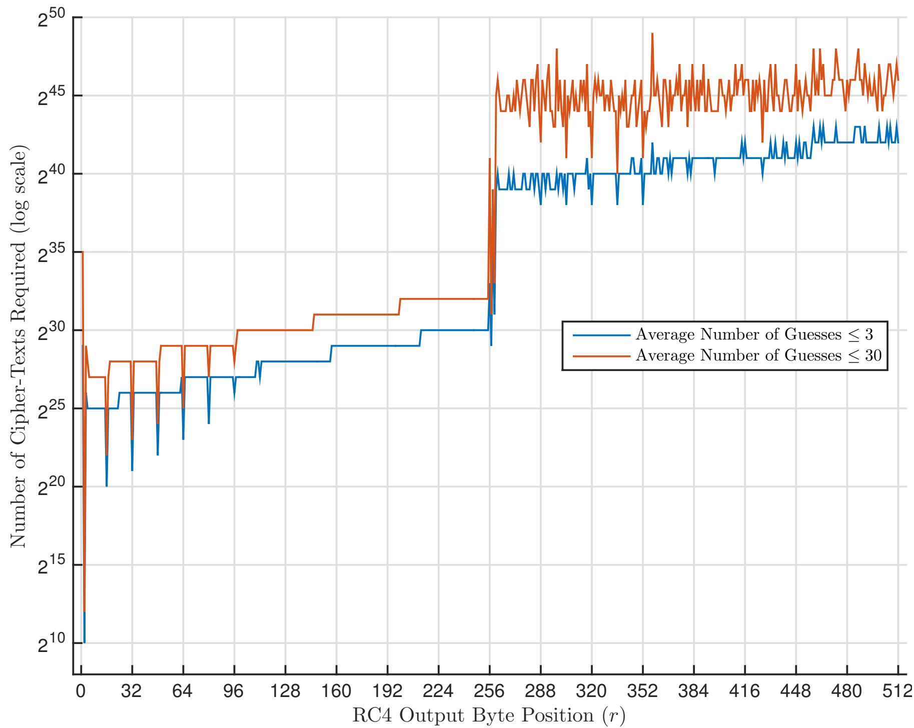
Third, the relative weakness of the "max-peak" attack is more costly in the second page than the first. This is apparent not just from how many more ciphertexts are required on average for the second page under the "Max-Peak" attack, but also from the wide variation in the figures for the second page under the "Max-Peak" attack as opposed to the relative consistency seen in the Bayesian attack. This means that to decipher the whole second page in the 1-shot setting requires over 2^{60} ciphertexts compared to roughly 2^{40} for the Bayesian attack. This is likely a result on the structural weakness in the "Max-Peak" attack that we described earlier in that it solely looks for candidates for the maximum peak. In the second page, where the distributions are much more uniform, there are many more candidates for the maximum that are close together resulting in more frequent failures with this attack.

Fourth, we can see clearly that the key-length related biases at positions $r = 16, 32, 48, 64, 80,$ and 96 and the strong bias towards 0 at $r = 2$ provide significant vulnerabilities relative to the other positions.

This concludes our report of the observations made as a result of our experiments. We now proceed to the final chapter of our report, detailing the conclusions we have drawn from our work.

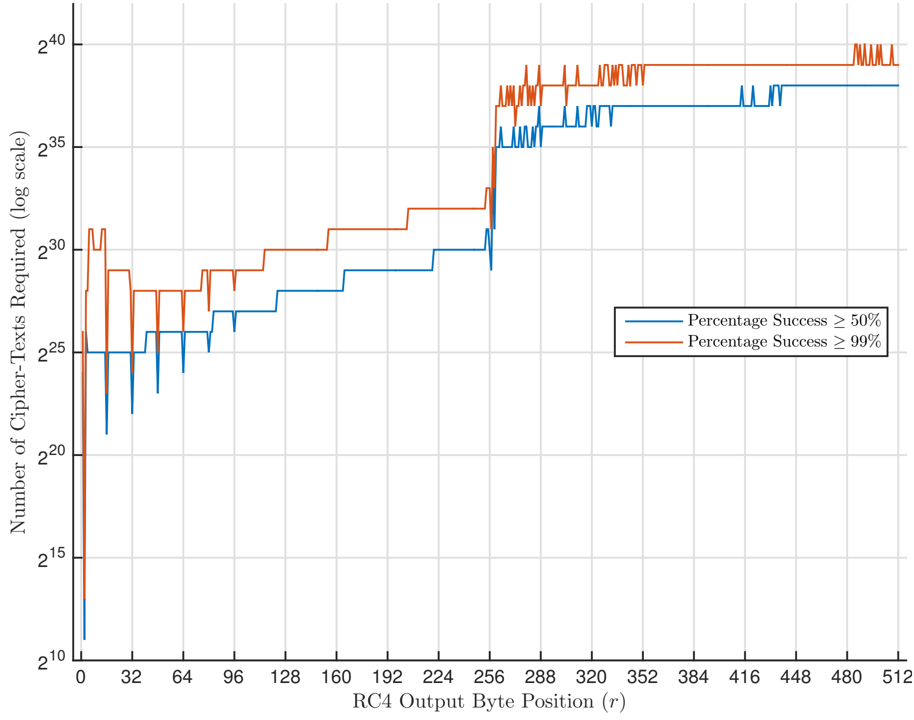


(a) Bayesian Attack

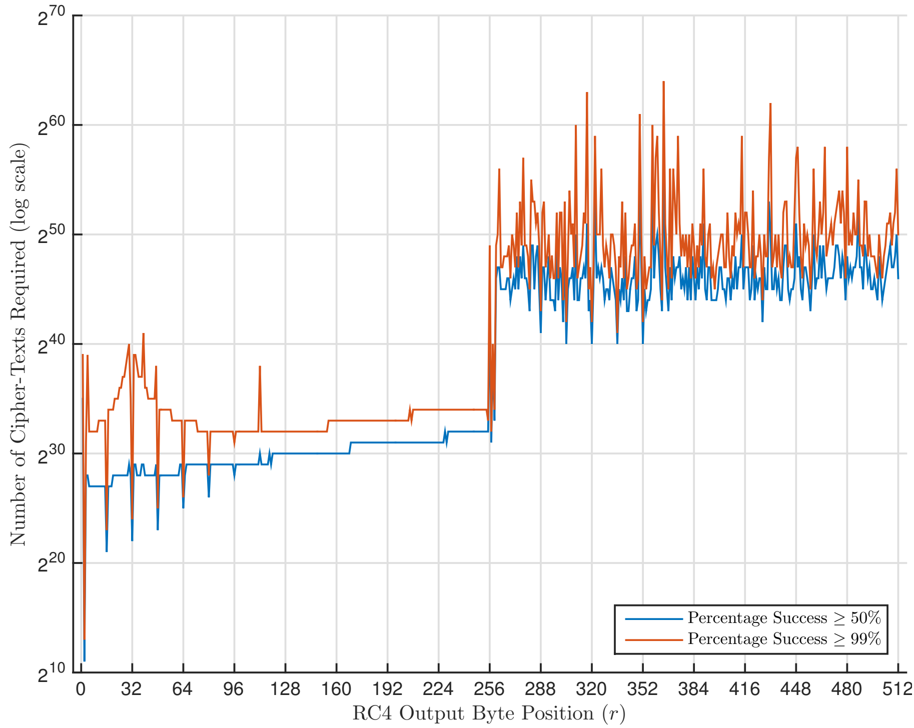


(b) “Max-Peak” Attack

Figure 4.14: The minimum (integral power of 2) number of ciphertexts required to reduce the average number of guesses under each of our n-shot attacks to below 3 and 30 for byte positions $1 \leq r \leq 512$.



(a) Bayesian Attack



(b) "Max-Peak" Attack

Figure 4.15: The minimum (integral power of 2) number of ciphertexts required to get the success rate of our 1-shot attacks over 50% and 99% for byte positions $1 \leq r \leq 512$.

Chapter 5

Conclusions

5.1 Scientific Conclusions

5.1.1 Confirmed and Unconfirmed Biases

The data we have collected confirms the existence and scale of most of the biases reported in the work of Al Fardan et al. [3] as well as Isobe et al. [8]. This includes many of the most significant biases in the first page of the RC4 key-stream and the Fluhrer-McGrew biases. However, there are several discrepancies which we must point out.

First, the biases b_0^r do not all fit the pattern predicted in the literature found in Gupta et al. [14] and, whilst the data in [3] and [8] appear to be consistent with our observations, they do not explicitly mention the deviations from these predictions. These deviations were found at the values $r = 4, 5$, $r = 31, 32, 33$, $r = 34, 35$, and $r = 64, 65$ where we see runs of increasing biases instead of the predicted decreasing biases. The theoretical prediction of these bias values in [14] is the result of an approximation so it is possible that inaccuracy in this calculation is the source of the observed discrepancy. However, it seems more likely that it is the result of some key length dependent contributions to the biases which are not considered explicitly as part of the theoretical calculations. We have seen other key-length dependent biases and the r values producing the discrepancies are local to even multiples of the key length, 16.

Second, contrary to [3], the biases b_r^r are not always positive for $1 \leq r \leq 256$. They are in fact negative for $r = 1, 2, 256$. This appears to be in accordance with the figures presented in [8].

Third, the biases b_{256-r}^r are not positive for all r which are multiples of 16 on the first page as appears to be proposed in [3]. This, in fact only holds for $r \leq 128$. These biases for $128 < r \leq 256$ are not explicitly discussed in [8].

5.1.2 Discovered Biases

We have discovered new single byte biases on the second page of the RC4 key-stream. The largest discoveries are a factor of 10 smaller than the largest biases on the first page but they are prominent never-the less. The most significant biases are as follows:

- The first three byte positions of the second page, $r = 257, 258, 259$, have biases towards $Z_r = 0, 2, 3$ respectively.
- For $260 \leq r \leq 384$, our data suggests that the values $b^r_{r+1 \bmod 256}$ are negative and increasing, although the upper limit on r may be slightly lower when they turn positive. These biases manifest at the leading edge of a sequence of biases b^r_a for $0 \leq a \leq r$ which are much closer to 0.
- For $r \equiv 0 \bmod 16$ and $257 \leq r < 384$, $b^r_{2r \bmod 256}$ is positive. Whilst we were completing our work this bias was independently discovered and published by Vanhoef and Piessens [16].
- Beyond the above there appears to be a general trend of biases away from lower byte values and towards higher ones for the second page. This trend peaks at around $a = 192$ before the biases begin to decrease again. This trend gets less prominent as we move through the second page and, with our data, is essentially unverifiable by the end of the page.

5.1.3 Measuring Vulnerability

Entropy is a well justified theoretical measure of non-uniformity in a probability distribution. The lost entropy in the key-stream output at a particular byte position would give us precisely the amount of information gained about the message byte value from one ciphertext. In theory then, entropy is an ideal measure of vulnerability to ciphertext only attacks. Unfortunately, we have found that it is not effective in our setting of a broadcast attack on single bytes of RC4 output. With the data we were able to sample from RC4 key-stream outputs estimating the entropy doesn't even allow us to distinguish the key-stream distributions from uniform. An attacker who only took this information into account might therefore conclude that COAs on RC4 would not be possible.

However, our other metrics of vulnerability imply that this would be far from true. Our other measures show that an attacker can in many cases gain great confidence of the message bytes with potentially practicable numbers of ciphertexts, especially on the first page. This suggest that the key-stream single byte entropy loss can be misleading as a measure of vulnerability if blindly trusted.

Whilst our work has focused on particular metrics of vulnerability, we have not had the time to pursue the practical implementation of our attacks and consider their danger in the wild. Al Fardan et al. [3] implemented attacks to recover plaintexts from HTTPS traffic using both single and double byte biases. They found the ciphertext requirements for the single byte attack to be in accordance with our results that 2^{32} ciphertexts should be sufficient to get high confidence for all of the first page bytes. However, recovering a HTTPS cookie required significant amounts of time (c. 2000 hours) and bandwidth that was not clearly feasible for practical exploitation. This work was followed by Garman et al. [7] who further developed the Bayesian attack scheme to take account of non-uniform priors over the message space. This is a plausible assumption, particularly when the plaintext is restricted to human readable text. In parallel, the recent work by Vanhoef and Piessens [16] has sought to improve on the Al Fardan et al. attack by simultaneously exploiting various multi-byte biases with the single byte biases. This attack was able to recover HTTPS cookies in 52 hours, a significant improvement from earlier work. We can anticipate that using this simultaneous exploitation of different biases together with non-uniform priors would further

improve on the running time of potential attacks.

At this time then, the vulnerabilities of RC4 are within reach of practical exploitation and with increasing hardware speed this will of course get worse.

Our measures of vulnerability could prove useful in considering the potential weaknesses in other stream-ciphers. Any stream-cipher producing similar values to RC4 might not be immediately breakable in practical time spans. However, if it were not immediately discarded out of precaution, it would still merit further investigation for other biases before being adopted for any high-value use.

5.1.4 Potential Future Extensions

In terms of bias discovery we hypothesise that there will be single byte biases on the third page of the RC4 key-stream but that they will be smaller again than the second page. Whether these biases might be usefully exploited would be worth exploring.

With sufficient resources we would have liked to collect sufficient data to resolve the Fluhrer-McGrew biases to a greater degree of accuracy. We would be interested to test the accuracy of the predicted values further and this is an obvious area for extension.

We have already mentioned the potential for attacks simultaneously exploiting single and multi-byte biases as well utilising non-uniform priors over the message space. This may involve the use of more involved analytical tools such as Markov Chain modelling for the computation of posterior distributions over the message space which may involve higher running times. The feasibility of such an attack would be worth investigating further.

5.2 Other Insights Gained

5.2.1 Coding for Speed

Our work required us to produce significant amounts of RC4 output. As a result we had to optimise our code for speed. As well as learning many techniques for achieving this the author also came to appreciate the amount of effort that can be saved by following the adage of Knuth, "*premature optimization is the root of all evil.*"

5.2.2 Continued Use of RC4

Based on our results and the practical exploitation performed by others including [8, 3, 7, 16], we join in strongly urging that RC4 no longer be used. If the published attacks or similar schemes have not already been implemented in the wild then we should not expect our luck to hold. The resources required to carry out these attacks are within reach for non-state actors, and ignorance is no longer a viable excuse for inaction.

5.2.3 Lessons for Crypto System Selection

Finally, we feel it is sensible to consider some wider lessons for crypto system selection based on the history of RC4.

It is worth asking why RC4 became so widely used in the first place. Was it because it was easy to implement and relatively fast? Because it was an RSA commercial product and people trusted the source? Or perhaps, because no one knew a way to brake it? We are not in a position to know for sure but it seems plausible that a combination of these factors helped. From a security perspective these factors are at best proxies for the central question of how vulnerable a cipher is to attack. However, they may have been the best available evidence on which to base a decision. In the early analysis of RC4 it seems that the broadcast attack setting was not even considered. This is perhaps understandable based on the fact that a typical use cases of the mid-nineties would make the most recent attacks completely impractical. However, we should consider why it took until recently to discover the multiplicity of biases in RC4 given that some initial significant biases were already discovered in 2001? Was it a paucity of attack models, a lack of computing or a lack of interest?

The history of RC4 indicates that cipher selection has on occasion taken place based on imperfect/incomplete information and, despite the impressive efforts of cryptanalysts, the available resources have not been sufficient to fill this gap. This is not to say that further resources could not be made available. However, there are ongoing barriers to this. In general private companies suffer disincentives to committing resources towards publishing cryptanalysis work because the benefits of the work are shared with all and sundry, leading to a tragedy of the commons situation. Governments have large incentives towards getting cipher selection right, especially as many public protocols form parts of critical national infrastructure. However, they also have conflicts of interest. Carrying out surveillance for national security reasons is seen by most governments as a requirement, therefore, their cryptanalytic output is not always trusted.[1] Finally, non-governmental organisations, who have lead the design of public protocols to date, generally do not have the spare resources to increase focus on cryptanalysis.

The problems that the story of RC4 highlights are therefore not easy to solve. However, we hope that the ubiquity of RC4 combined with its growing tapestry of weaknesses should lead to some further reflection about the process for cipher selection in protocol design.

[1] Aspects of the Snowden documents relating to NSA's contribution to the design of Dual EC DRBG have further damaged trust in this regard.

Acknowledgements

The author would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. http://dx.doi.org/10.5281/zenodo.22558.

I would also like to thank Prof. Kenny Paterson of Royal Holloway for his kind help.

Finally, I would like to thank my project supervisor, Prof. Andrew Ker, for his ever helpful and insightful guidance throughout this project.

Bibliography

[1] 802.11-1997 - IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1997.

[2] 802.11i-2004 - IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 6: Medium Access Control (MAC) Security Enhancements, 2004.

[3] 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.

[4] Thai Duong and Juliano Rizzo. *Here Come the ⊕ Ninjas*. Ekoparty, 2011.

[5] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In Serge Vaudenay and AmrM. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2001.

[6] Scott R. Fluhrer and David A. McGrew. Statistical Analysis of the Alleged RC4 Keystream Generator. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bruce Schneier, editors, *Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 19–30. Springer Berlin Heidelberg, 2001.

[7] Christina Garman, Kenny Paterson, and Thyla van der Merwe. Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS. Technical report, Johns Hopkins University and Royal Holloway, 2015.

[8] Takanori Isobe, Toshihiro Ohigashi, Yuhei Watanabe, and Masakatu Morii. Full Plaintext Recovery Attack on Broadcast RC4. In *Proc. the 20th International Workshop on Fast Software Encryption*, 2013.

[9] Itsik Mantin. Predicting and Distinguishing Attacks on RC4 Keystream Generator. In Ronald Cramer, editor, *Advances in Cryptology EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 491–506. Springer Berlin Heidelberg, 2005.

[10] Itsik Mantin and Adi Shamir. A Practical Attack on Broadcast RC4. In *PROC. OF FSE 01*, pages 152–164. Springer-Verlag, 2001.

[11] Toshihiro Ohigashi, Takanori Isobe, Yuhei Watanabe, and Masakatu Morii. How to Recover Any Byte of Plaintext on RC4. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 155–173, 2013.

[12] Santanu Sarkar, Sourav Sen Gupta, Goutam Paul, and Subhamoy Maitra. Proving TLS-attack related open biases of RC4. *Designs, Codes and Cryptography*, pages 1–23, 2014.

[13] Bruce Schneier and Doug Whiting. Fast software encryption: Designing encryption algorithms for optimal software speed on the intel pentium processor. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 242–259. Springer Berlin Heidelberg, 1997.

[14] Sourav Sen Gupta, Subhamoy Maitra, Goutam Paul, and Santanu Sarkar. (Non-)Random Sequences from (Non-)Random Permutations - Analysis of RC4 Stream Cipher. *Journal of Cryptology*, 27(1):67–108, 2013.

[15] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. In *NDSS*. The Internet Society, 2002.

[16] Mathy Vanhoef and Frank Piessens. All Your Biases Belong To Us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security 2015*, 2015.